Universidad de Alcalá
Escuela Politécnica Superior

Ingeniería Electrónica y Automática Industrial

Trabajo Fin de Grado

Modelado y control de la mano robótica Inmoov-SR y brazo robot IRB120 en MATLAB y V-REP

Autor: Miguel Encabo Fabián

Tutor/es: Elena López Guillén

2017

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Ingeniería Electrónica y Automática Industrial

Trabajo Fin de Grado

Modelado y control de la mano robótica Inmoov-SR y brazo robot IRB 120 en Matlab y V-REP

Autor: Miguel Encabo Fabián

Tutor/es: Elena López Guillén

TRIBUNAL:

Presidente: Alejandro Martínez Arribas

Vocal 1º: Ernesto Martín Gorostiza

Vocal 2º: Elena López Guillén

FECHA: 21/09/2017

Quiero dedicar este trabajo a todos aquellos
que me han apoyado para llegar a donde estoy.

A mi familia, a los que están y a los que ya no, pero
que gracias a todos ellos soy el chico que soy.

Y a todos mis amigos, a los que conozco desde
pequeño, y a aquellos que me han acompañado
durante estos cinco años. Empezaron como
desconocidos y han terminado siendo parte de mí,
apoyándonos durante todo este tiempo para
conseguir estar aquí y lo que está por venir.

Índice

Índice de Ilustraciones

Resumen

En este proyecto se va a estudiar el proceso de diseño y simulación de la herramienta InMoovHand-SR conectada a un brazo robótico mediante las herramientas V-REP y MATLAB, a través de las cuales se programará el transcurso de la simulación y se podrá observar el comportamiento real de una estación de trabajo.

Se ha fijado como objetivo principal la realización de simulaciones dentro de un entorno virtual de posibles situaciones reales con el fin de obtener un control óptimo de la herramienta, así como sopesar las diferentes funcionalidades de la misma.

Abstract.

This project is going to study the design and simulation process of the InMoovHand-SR tool connected to a robotic arm throughout the V-REP and MATLAB tools, by which the simulation course will be programmed and the actual performance of a workstation could be appreciable.

It has set as main target the simulations' implementation of possible real situations, within a virtual environment, in order to obtain an optimal monitoring of the tool, as well as to weigh its different functionalities.

Palabras claves / Key Words

- MATLAB
- VREP
- InMoov-Hand
- IRB120
- Toolbox de Peter Corke

Resumen extendido

La idea para la realización de este proyecto surgió al encontrar en internet la mano Inmoov la cual fue diseñada y subida a internet con el fin de que cualquiera pudiera construir dicha herramienta mediante una impresora 3D y unos simples pasos que seguir.

En este caso concreto se decidió que sería de gran utilidad esta herramienta junto a un brazo robótico, pudiendo simular los movimientos completos de un brazo humano. Pero para ello sería necesario, además del montaje en si, un sistema de control de las articulaciones. Estos sistemas son complejos de manejar y por lo cual son necesario el uso de software especializado para un correcto funcionamiento.

Como es evidente, un proceso como este debe pasar por muchas fases previas al funcionamiento completo del sistema, y una de ellas es la programación y simulación de resultados en un entorno virtual, esta parte va a ser la desarrollada en el presente proyecto con la ayuda de V-REP y MATLAB.

El encargado principal de la simulación es el software V-REP en el cual deberemos introducir tanto el brazo robótico a utilizar, IRB120 (ABB), como la herramienta en nuestro caso la mano Inmoov. Como ambos modelos no se encuentran entre la amplia gama de robots y herramientas que nos ofrece el simulador deberemos generarlos para poder trabajar con ellos. Es imprescindible mencionar que debido a la necesidad de requisitos que requiere la simulación, el eje principal del proyecto girara entorno a este software siendo él el que realice la mayor parte de las operaciones gracias a la ejecución de secuencias de código que se desarrollaran para realizar el objetivo fijado.

En el caso de la herramienta MATLAB el modelado del sistema es plenamente esquemático, consiguiendo comprobar desde este software los movimientos que se pueden realizar por medio de la mano robótica y el brazo IRB120. Esto es debido a que MATLAB no es una herramienta enfocada principalmente a la robótica, pero gracias a la Toolbox compartida por Peter Corke, podemos obtener resultado mediante el modelado y simulación de sistemas robotizados de una complejidad no del todo elevada.

La utilidad principal que se le va a dar a MATLAB en el desarrollo de esta investigación es principalmente la de comunicación y transferencia de información con el entorno de simulación de V-REP, pudiendo de esta forma enviar una posición determinada al robot u obtenerla, así como de cualquier objeto presente en la escena simulada. Para esto V-REP posee un protocolo de comunicación API, el cual se utiliza para generar la conexión, el cual contiene un gran catálogo de funciones tanto para interactuar entre las dos plataformas como para programar la simulación en el entorno de V-REP. También se han utilizado con este fin una variedad de funciones dadas en la Toolbox de Peter Corke.

Memoria

1. Introducción

La robótica es la rama de la ingeniería mecatrónica, de la ingeniería eléctrica, de la ingeniería electrónica, de la ingeniería mecánica, de la ingeniería biomédica y de las ciencias de la computación que se ocupa del diseño, construcción, operación, disposición estructural, manufactura y aplicación de los robots.

La robótica combina diversas disciplinas como son: la mecánica, la electrónica, la informática, la inteligencia artificial, la ingeniería de control y la física. Otras áreas importantes en robótica son el álgebra, los autómatas programables, la animatrónica y las máquinas de estados. [1]

Sus aplicaciones son de lo más extensas en la actualidad, desde el ámbito doméstico hasta el industrial pasando por tales como la medicina e incluso la docencia, ya que en la actualidad se están implantando sistemas de inteligencia artificial y robóticos en prácticamente todos los lugares.

Este proyecto combina la utilidad de un brazo robótico industrial fabricado por la empresa ABB con una herramienta en forma de mano humana queriendo aproximarnos así a la biotecnología, pudiendo obtener muy diversas posibilidades, como sería la manipulación de objetos de tamaños y pesos considerados como grandes hasta la manipulación de piezas mucho más pequeñas, con alta exactitud en los movimientos.

El objetivo principal que se aborda en este trabajo es el modelado y control tanto del brazo robótico IRB120 como de la herramienta Inmoov-SR en un entorno virtual el cual nos permita realizar simulaciones de comportamientos reales, mediante la comunicación entre los entornos de MATLAB y V-REP.

2. Objetivos y método de desarrollo.

El objetivo principal marcado para este proyecto, como ya se ha explicado con anterioridad, es el control de la mano robótica en un entorno de simulación virtual. Como todo objetivo final está compuesto por diferentes subprocesos a través de los cuales terminamos desarrollando el conjunto de la aplicación.

Los subprocesos que se han aplicado como fases del trabajo son los que siguen:

- **Estudio previo y preparación de herramienta**

  En primer lugar, se llevó a cabo un proceso de investigación y estudio de la herramienta Inmoov-SR previo a la realización del proyecto, obteniendo de esta forma los datos necesarios para comprobar la compatibilidad con el brazo robótico que disponemos y la posibilidad de fabricar la herramienta por nosotros mismos.

  Como el resultado del estudio fue favorable se obtuvieron los ficheros necesarios para la impresión en 3D de la herramienta, así como la guia de montaje de la misma. Esto fue descargado a través de las siguientes páginas web: http://inmoov.fr/hand-and-forarm/ y https://www.thingiverse.com/thing:18939 .

- **Montaje de la herramienta.**

  Este subproceso engloba todas las acciones en las cuales se obtuvieron mediante la impresora 3D las piezas de la mano, así como su montaje y la fabricación de una pieza adicional que sustituyera a la muñeca facilitada para una mejor conexión al IRB120.

  La fabricación de la muñeca se realizó gracias al software de diseño gráfico SolidWorks a través del cual se obtuvo el fichero necesario para la impresión.

- **Instalación del software requerido.**

  En este caso el proceso es sencillo ya que solo es necesario para el desarrollo de este proyecto MATLAB, la toolbox de robótica de Peter Corke y el simulador V-REP**.**

- **Estudio del software a utilizar y de sus utilidades.**

  Obtención mediante el uso y documentos auxiliares del software de las nociones necesarias para desarrollar el trabajo. Cabe destacar la complejidad de los dos softwares utilizados siendo muy amplias sus posibilidades y por tanto difíciles de utilizar a la perfección.

- **Modelado del brazo robótico ABB IRB120 en el simulador de V-REP.**
- **Modelado de la herramienta Inmoov-SR Hand en el simulador V-REP.**
- **Control del ABB IRB120 y establecimiento de conexión ente MATLAB y V-REP.**
- **Control de la herramienta Inmoov-SR.**

2.1 Esquema funcional.

En la siguiente imagen se muestra mediante un diagrama de bloques los objetivos y procesos seguidos para el fin buscado en este proyecto. Se refleja el esquema de funcionamiento del trabajo separado por el software donde se realiza cada función.

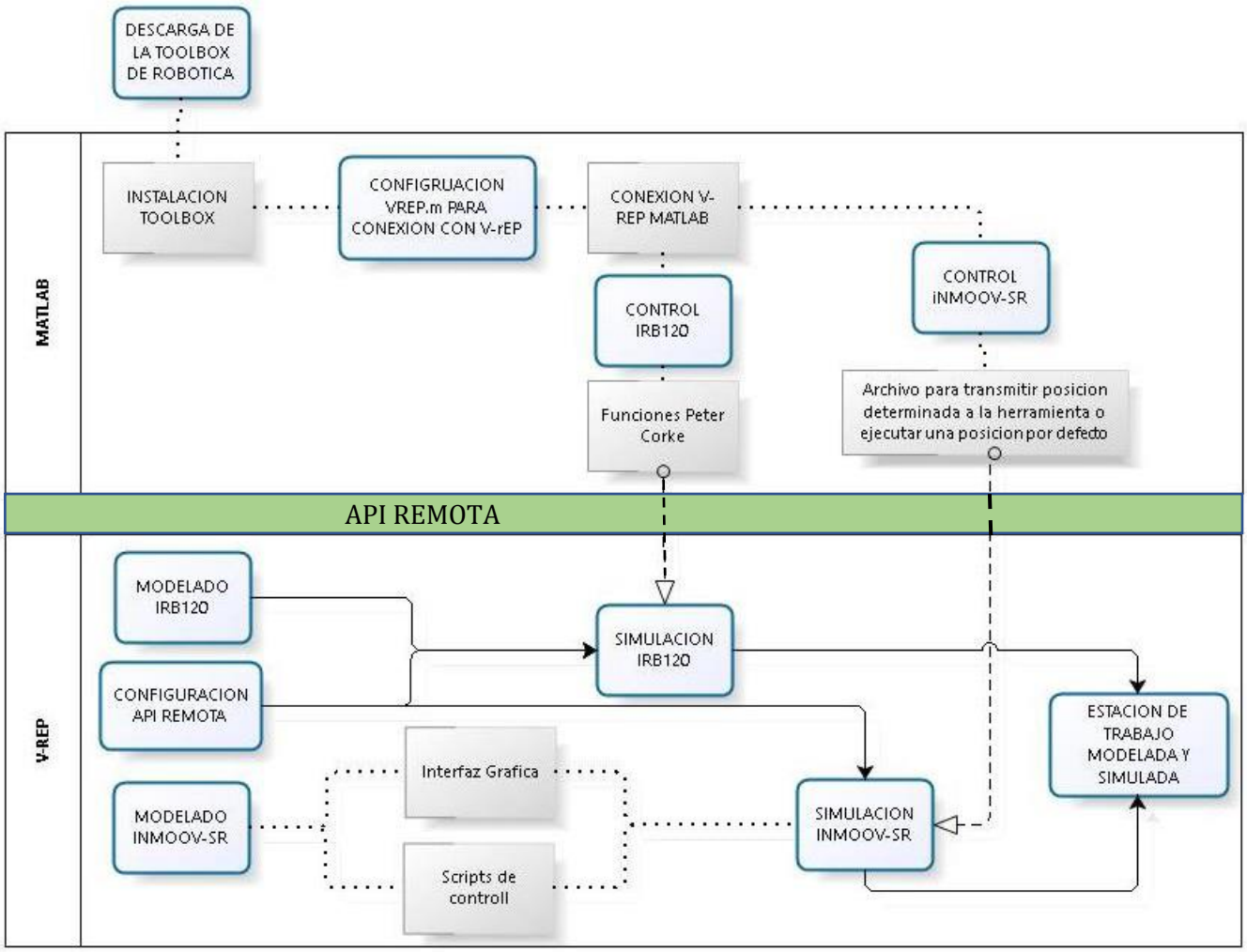

Ilustración 1, Esquema Funcional

3. Arquitectura del proyecto.

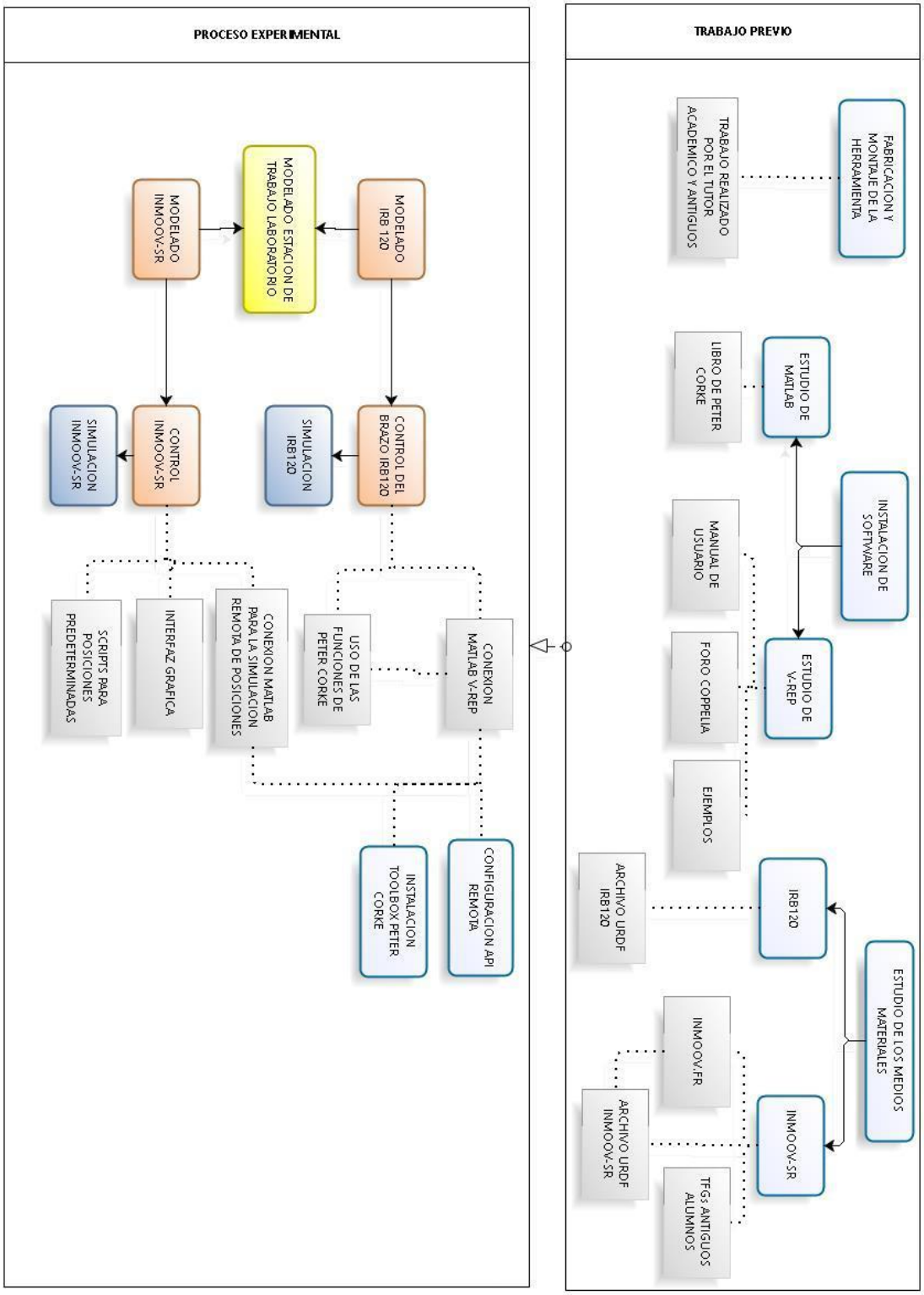

Ilustración 2, Arquitectura del proyecto

4. Simulador V-REP

Este entorno de simulación virtual ha sido elegido para el desarrollo de este proyecto gracias a las amplias posibilidades que ofrece, así como sus altas características dentro de este tipo de software. En este apartado se va a tratar de explicar el funcionamiento del software con toda la variedad de posibilidades que nos aporta y que nos han sido de utilidad, dejando de lado las características más avanzadas en las cuales no se ha llegado a hondar en este trabajo.

Las principales razones por las que ha sido seleccionado son las siguientes:

- Se ofrecen actualizaciones frecuentemente mejorando la estabilidad del sistema.
- Este simulador se ofrece para todos los sistemas operativos utilizados en la actualidad, ofreciéndonos de esta forma facilidades de compatibilidad para todos los interesados en el proyecto.
- Gran información sobre sus características, así como guias de uso en diversas páginas web [2] y foros [3].
- Versión completa gratuita para el sector educativo.
- Importación rápida de modelos en 3D desde software específico, facilitando el trabajo de modelado.
- Posibilidad de comunicación en muchos lenguajes de programación, lo que nos es de utilidad para la utilización remota e interacción mediante Matlab.

Los desarrolladores de este entorno virtual lo elevan como el simulador más completo y con mayor número de funciones en el mundo de la robótica, de esto viene dado que su uso sea complejo y costoso de aprender sin una base previa en este tipo de lenguaje.

Las características y posibilidades de este simulador son:

- Mas de 400 funciones.
- Simulación por 4 motores. (ODE, Bullet, Vortex, Newton)
- Cálculos aritméticos de distancias entre objetos.
- Detección de colisiones.
- Calculo de cinemática directa e inversa.
- Sensores de visión, de fuerza, de proximidad, etc.
- Cálculos de trayectorias.
- Gran variedad de modelos preinstalados.
- Escenas ya creadas a modo de ejemplo.
- Interfaz de usuario personalizados
- 7 lenguajes de programación (C/C++, Python, Java, Lua, Matlab, Octave o Urbi)
- Registro de datos y visualización (gráficos de tiempo, gráfico X / Y o curvas 3D)
- Función multiniveles deshacer /rehacer (undo/redo), función de grabación o registro de video, simulación de pintura, documentación exhaustiva, etc.

4.1 Interfaz de usuario

En este aparatado se va a abordar la distribución y posibilidades que nos ofrecen en la interfaz gráfica dentro del entorno de simulación virtual V-REP.

Se va a hondar en las características que han sido de mayor utilidad para el desarrollo de este proyecto, quedando el resto y las mismas explicadas de forma más amplia en el manual de usuario que nos proporciona de forma online Coppelia. [2]

En siguiente imagen se ve la interfaz gráfica que se ofrece al usuario:

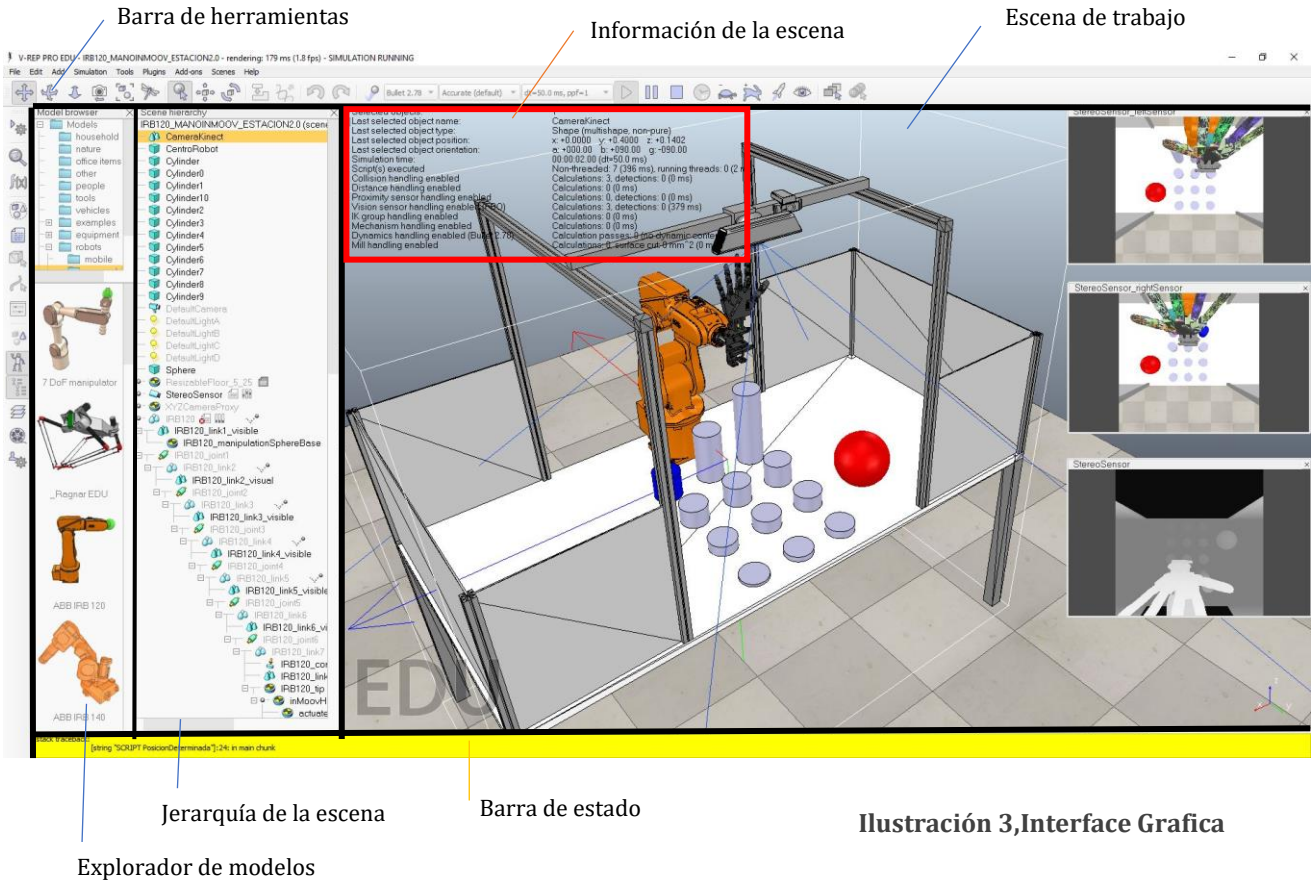

Ilustración 3,Interface Grafica

- **Explorador de modelos.**

En esta ventana se nos ofrecen todos los modelos predeterminados que incluye la instalación del software, así como los modelos que el usuario cree y guarde en el path correspondiente a los modelos del programa.
Estos modelos incluyen desde robots, tanto móviles como fijos, hasta sensores de movimiento, cámaras 3D y objetos inanimados, pudiendo ser plantas simulación de personas mobiliario…

- **Barra de estado.**

En ella el programa nos notifica las acciones que se van ejecutando, informa de los errores o problemas que se pudieran producir en la simulación de alguna escena o de algún script.

Se usa frecuentemente, debido a la falta de un depurador, para mostrar mensajes en determinados puntos del código para obtener fiabilidad de su funcionamiento.

- **Barra de herramientas.**

En la barra de herramientas podemos encontrar todas las funcionalidades que son de mayor utilidad tanto para el modelado de escenas como para su simulación, a continuación, se muestran los iconos con las acciones que realiza cada uno.

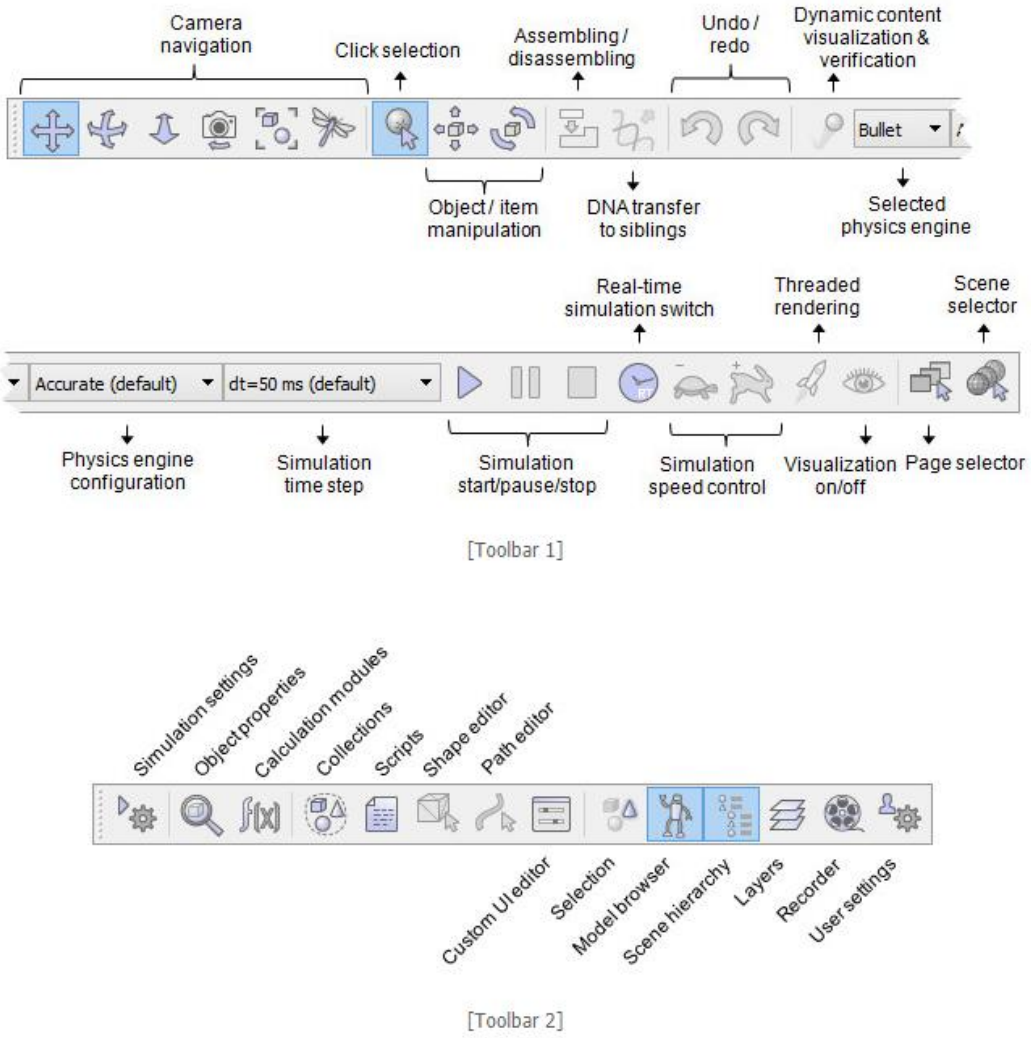

Ilustración 4, Barra de herramientas [2]

- **Escena de trabajo.**

En esta ventana se lleva a cabo el diseño y simulación de estaciones de trabajo pudiendo observar su comportamiento y recrear situaciones que podrían acontecer en la realidad para asegurarnos un correcto funcionamiento.

Además, en esta ventana pueden aparecer otras sub-ventanas previamente configuradas que muestren datos de sensores, interfaces personalizadas o simplemente otras cámaras para la correcta visualización de la simulación.

- **Jerarquía de la escena**

En esta ventana se muestran todos los objetos que existen dentro de la escena creada, así como las diferentes escenas abiertas para cambiar rápidamente entre ellas.

Los objetos de una escena pueden aparecer de forma individual u ordenadas en forma de árbol, esta última ordenación suele componer robots más complejos manteniendo los objetos conectados y con una jerarquía de unos sobre los otros. Pero también se pueden juntar objetos sin necesidad de formar un robot, únicamente para tenerlos agrupados. A continuación, podemos ver una ilustración de esta ventana como ejemplo.

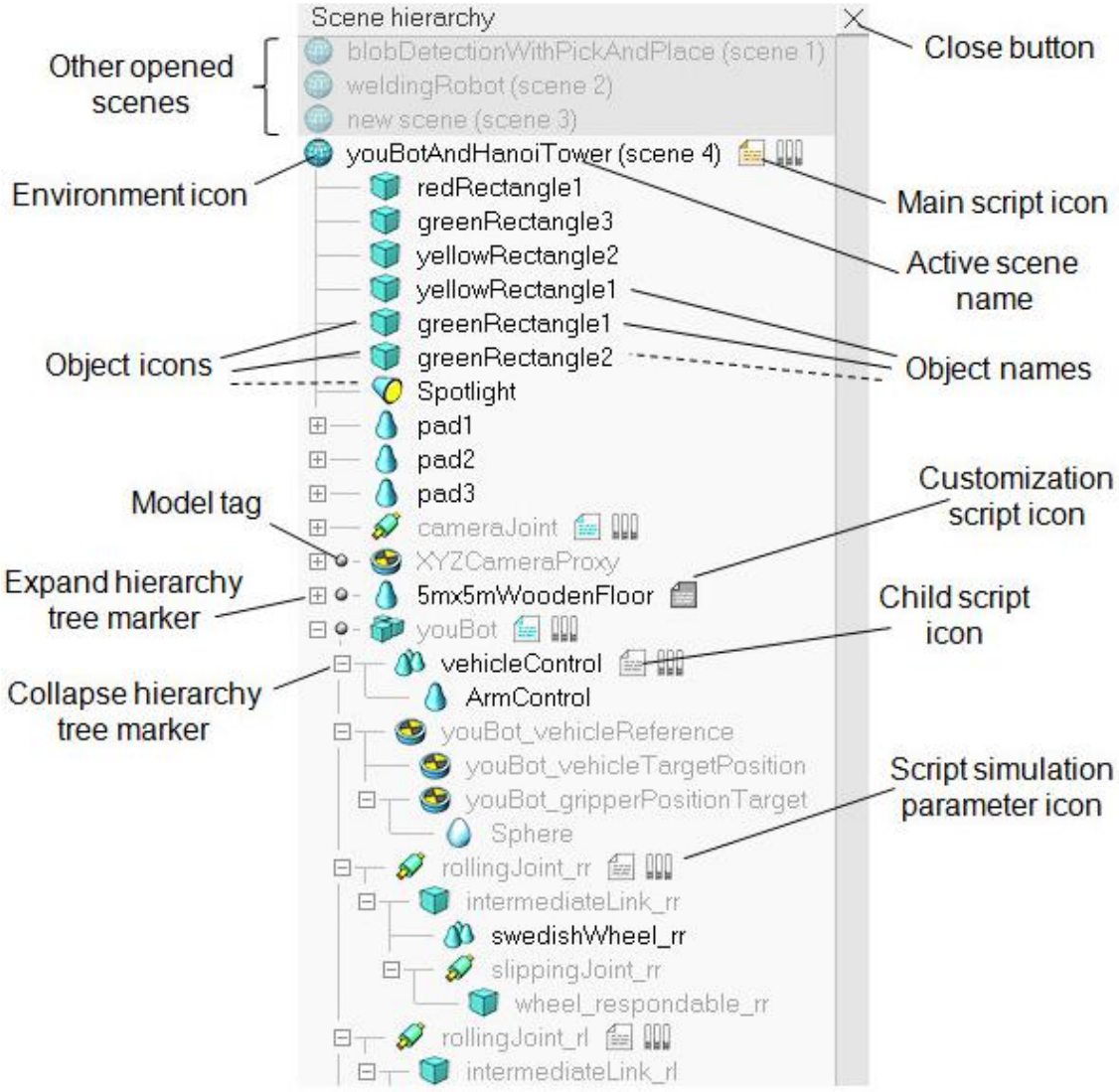

Ilustración 5, Jerarquía de la escena [2]

- **Información de la escena.**

En este display se muestra la información actual de un objeto seleccionado, así como los parámetros correspondientes a la simulación, siempre y cuando esta se esté ejecutando.

- **Interfaz grafico personalizado.**

V-REP ofrece la posibilidad de diseñar interfaces graficas personalidades (UIs) mediante las cuales se puedan manejar diferentes parámetros de la simulación a través de un script en el cual se configura las acciones a realizar al pulsar los diferentes botones creados en el interfaz o deslizando los sliders.

Un ejemplo de UI es el que se ha creado para realizar un control manual de la herramienta Inmoov-SR:

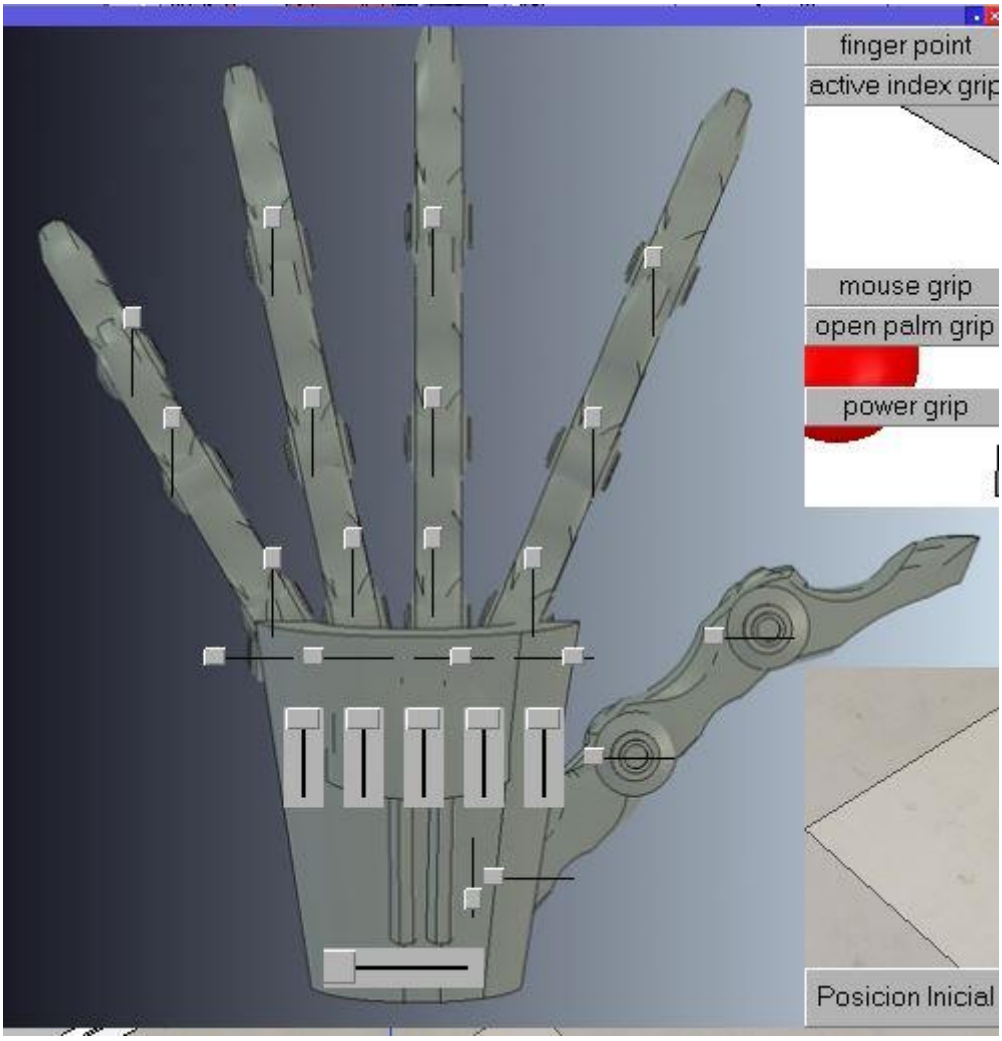

Ilustración 6, Interfaz de usuario personalizado

4.2 Formas

Las formas son uno de los objetos de V-REP al cual se le da más uso, esto se debe a que con ellas se generan todos los modelos de robots que posteriormente se usaran en las simulaciones. También forman parta de todos los objetos inanimados que se puedan utilizar como simulación de objetos reales.

Aunque las formas tienen un peso importante para la generación de robots, este software no suele ser usado para este fin, ya que es un trabajo muy laborioso para robots de cierta complejidad. Por ello V-REP dispone de múltiples métodos de importación de modelos creados en software de diseño 3D que facilitan mucho este proceso. En el caso que se contempla en este proyecto las formas se han usado como elementos auxiliares o únicamente se han tenido que modificar ligeramente en los modelos para obtener un resultado más preciso.

Cabe explicar, que al realizar las simulaciones con modelos generados mediante archivos .urdf se aplican ciertas restricciones en la simulación, ya que el programa lo toma como formas aleatorias no siendo capaz de realizar de forma totalmente fiable los cálculos de colisiones o de aplicaciones de fuerza.

Las principales formas que vamos a destacar son:

- **Simple Random Shape.**

Estas formas ilustran cualquier modelo, se caracterizan mediante la configuración de unas propiedades visuales tales como textura, geometría … Como se ha mencionado anteriormente estas formas ralentizan la simulación si se pretende utilizar la simulación de colisiones.

- **Compound Random Shape.**

Es el conjunto de las formas aleatoria simples, por lo que en un mismo grupo de objetos puede haber diferentes formas, texturas y colores.

- **Simple Convex Shape.**

Estas formas se diferencian de las anteriores en la optimización para el cálculo de colisiones, aunque para este fin son mejores las formas puras.

- **Compound convex shape.**

- **Pure simple shape**

Son formas primitivas de geometrías simples (cubos, cilindros, esferas …) Como se ha mencionado anteriormente y por recomendación de los diseñadores de Coppelia estas formas son las más eficientes para la simulación de colisiones y movimientos, ofreciéndonos una simulación muy fiable y con poco error.

- **Heightfield shape.**

Representa el suelo, se puede considerar una forma pura ya que simula una zona de reposo para el resto de los modelos o formas que conforman la escena.

4.3 Sensores

Este entorno de simulación consta de gran variedad de sensores, de los cuales los más importantes son:

- **Sensores de visión.**

Estos elementos obtienen las imágenes dentro de la escena obteniendo de ellas fotografías o mapas de datos los cuales se pueden usar mediante un programa externo para su análisis o la implementación de aplicaciones específicas como puede ser el seguimiento de objetos.

- **Sensores de proximidad.**

Estos sensores son muy útiles para robots móviles los cuales necesitan sabes los elementos que les rodean para no colisionar contra ellos. Nos ofrecen una detección de los objetos que se encuentran frente al sensor dependiendo del rango que le introduzcas, o simplemente almacenan la medición de distancia pudiendo el usuario usarla dentro de un script para diversas finalidades.

- **Sensores de fuerza.**

Son de los sensores que más se usan en robots y los que nos ofrecen una gran cantidad de finalidades. Son difíciles de usar debido a que dependen mucho de la forma del robot y de cómo este diseñado

En el proyecto que hemos abordado de no se ha profundizado mucho en el tema de los sensores, únicamente se han realizado pruebas sencillas al crear la estación de trabajo como de la que se dispone en el laboratorio con una cámara Kinect obteniendo la imagen de la misma.

Se ha planteado el uso de sensores de fuerza para simular el agarre de objetos de una forma ultra-realista ya que según la presión que se haga y el objeto que sea se cerrara la mano de una forma determinada. Pero finalmente esta aplicación la hemos dejado propuesta como trabajo futuro, debido a la complejidad y las modificaciones tan importantes que había que realizar tanto en los modelos como en los scripts de las simulaciones para conseguir unos resultados fiables.

4.4 Articulaciones.

Estos elementos son fundamentales a la hora del diseño de robots, a partir de este punto las nombraremos como joints, su nombre en inglés y su nomenclatura en V-REP.

Para el diseño de robots V-REP nos ofrece 3 tipos de articulaciones:

- **Prismáticas.**

Generan un movimiento lineal a lo largo del eje configurado para tal fin. Su uso generalmente es en robots manipuladores o de movimientos simples.

- **Rotacionales.**

Estas son desarrollan movimientos algo más complejos que las anteriores, que consisten en la rotación sobre un eje fijo. Tienen multitud de finalidades ya que nos ofrecen muchos movimientos. Simulan un motor rotacional que conectado a los eslabones del robot producimos los movimientos deseados. En muchos casos la combinación de varias articulaciones de este tipo sustituye al último tipo, las esféricas.

En nuestro caso todas las articulaciones que se han usado en el modelado tanto de la mano robótica como del brazo IRB120 son de este tipo.

- **Esféricas.**

Como se ha mencionado en el apartado anterior surgen de la combinación de 3 articulaciones rotacionales, obteniendo de esta forma 3 grados de libertad en un mismo punto.

Están diseñadas para el uso mediante los ángulos de Euler, configurando su posición fácilmente mediante los parámetros Alpha, beta y gamma.

Para la configuración de las articulaciones V-REP ofrece cinco modos de operación:

- Modo Pasivo
- Modo de cinemática inversa
- Modo de movimiento (Obsoleto)
- Modo dependiente
- Modo de fuerza o torque.

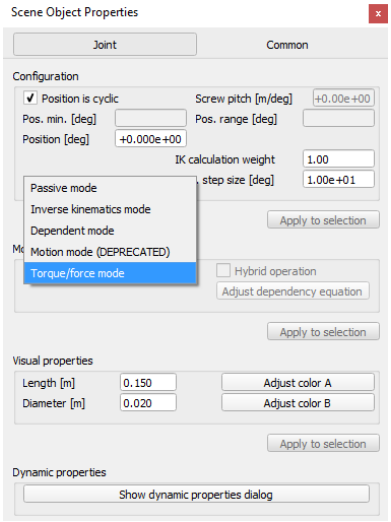

Ilustración 7, Propiedades de las joints

En este proyecto se va a usar el modo de cinemática inversa para el brazo robótico y el modo de torque para la mano robótica para que posteriormente en trabajos de ampliación se le puede implementar un sistema de sensores de fuerza para un mejor control.

4.5 Scripts

V-REP es un software muy avanzado también en el ámbito de programación, ya que nos ofrece múltiples métodos y lenguajes para el desarrollo de nuestro código. (plugin, API remotas, scripts, nodos ROS)

Para nuestra aplicación hemos utilizado tanto los scripts incrustados para la programación dentro de V-REP, como las funciones de API remota para la sincronización con MATLAB.

Hemos escogido trabajar con estos dos métodos ya que V-REP ofrece mucha información acerca de la programación en su lenguaje, lenguaje LUA, el cual se parece de cierto modo al C++, pero con una librería muy extensa de funciones específicas para el manejo de robots.

En este apartado se va a explicar cómo gestiona el software los diferentes tipos de script que se pueden programar.

En primer lugar, se encuentra el script principal, o main script, es el hilo de ejecución con mayor prioridad dentro de la simulación, ya que es el que la gestiona.

Este script se ejecuta cada vez que se produce un escalón en el tiempo de simulación o "time step" así como al principio de la simulación y al final de la misma, como se ve a continuación:

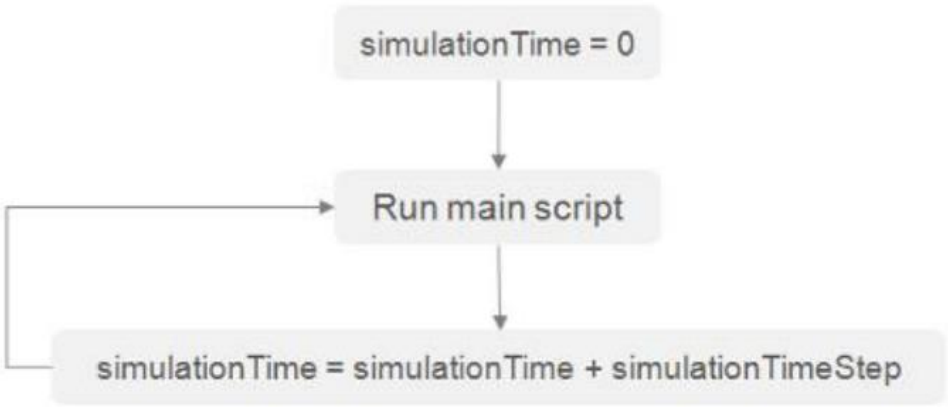

Ilustración 8, Bucle de simulación main script [2]

Dentro del script principal podemos observar 3 partes fundamentales:

- **Inicialización:**

  Es la parte del script que se ejecuta en primer lugar, es el encargado de introducir en la simulación todos los elementos que la componen.
  En esta parte se realizará el inicializado de la Api Remota, asignándole un puerto de conexión.

- **Parte sistemática.**

  Esta parte es la encargada de manejar todas las funciones de V-REP (detección de objetos, cálculos, datos de sensores…). En esta parte son fundamentales dos comandos "simLaunchThreadedChildScript" y "simHandleChildScript" ya que son los encargados de ejecutar los scripts secundarios o script hijos, tanto los hilados como los no hilados.
  A su vez esta parte se encuentra dividida en otras 3:
  - La zona de actuación, la cual se encarga de dar paso a los actuadores y sus diferentes usos dentro de los scripts secundarios.
  - La zona de sondeo, en esta parte se obtienen los datos generados por todo tipo de sensores desde los de visión hasta los de fuerza.
  - La zona de resultados se encarga de mostrar en la simulación como queda la escena después de las otras dos partes y las funciones de los scripts secundarios que se hayan ejecutado.

  Estas partes se ejecutan en el siguiente orden:

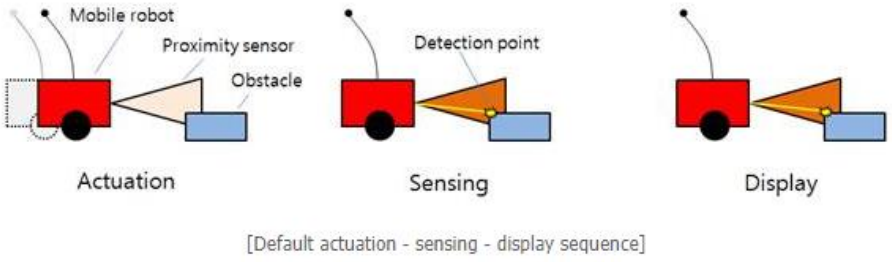

Ilustración 9, Secuencia main script [2]

- **Parte de restauración.**

  Esta parte se ejecuta una única vez antes de finalizar la simulación restableciendo todos los objetos a su posición inicial y borrando los datos innecesarios.

Vamos, por último, a describir los scripts hijos los cuales pueden ser de dos tipos:

- **Scripts hilados "threaded child scripts".**

  Estos scripts son ejecutados en un hilo diferente al de simulación, cuando uno de estos scripts es ejecutado no se puede volver a llamar hasta que no ha finalizado por completo. Estos scripts consumen muchos recursos y requieren de un gran tiempo de procesado.
  Han sido usados en este proyecto para la generación de los movimientos predeterminados.

- **Scripts no hilados "non-threaded child scripts".**
  Estos scripts son llamados por el script principal durante la fase de actuación y la de sondeo, debiendo devolver la ejecución al script principal cuando finalicen ya que sino la simulación produce un error. Operan como funciones para realizar un determinado proceso. En este proyecto se ha usado para crear el main script el cual llama a los scripts hilados.

El método más sencillo y el cual se va a usar para comunicarnos entre script son las señales "signals" a través de las cuales podemos transferir datos.

4.6 API REMOTA

En este proyecto la API remota tienen un papel fundamental, esto es debido a como ya se ha dicho y se explicara más adelante el control de la simulación se va a llevar a cabo mediante scripts incrustados y el uso de la API remota.

Este complemento nos permite programar de forma sencilla únicamente con MATLAB y V-REP, mientras que otros softwares que desarrollan funciones similares necesitan otros complementos. Tiene una función especial que nos permite controlar la simulación desde otro PC mediante la configuración del IP, pero esta opción no va a ser necesaria utilizarla en este trabajo.

La API remota está compuesta por una gran cantidad de funciones recogidas en el manual de usuario de V-REP [4] dichas funciones son utilizables desde otros programas en múltiples lenguajes: Matlab, C, C++, Python, Octave, Java…

La comunicación entre los programas se realiza en cualquiera de los anteriores lenguajes mediante una simulación síncrona o asíncrona.

En la siguiente imagen se observa la arquitectura cliente servidor creada por la API remota:

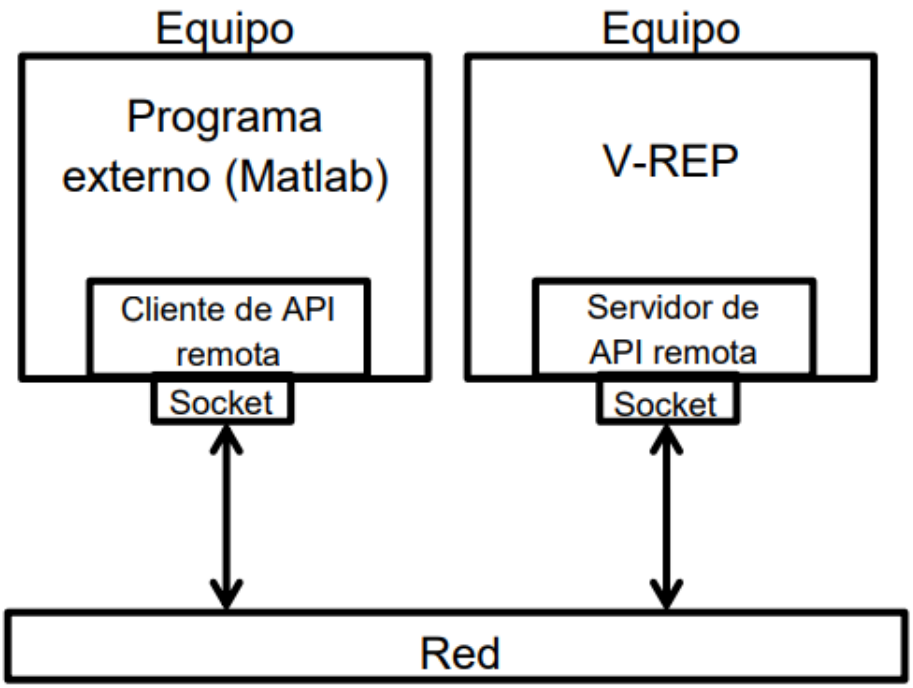

Ilustración 10, Arquitectura API remota

Explicado de forma sencilla la API remota establece una conexión entre el entorno de simulación virtual y un programa externo, en nuestro caso, MATLAB. Desde esta conexión se controla la simulación mediante las funciones correspondientes.

4.6.1 Servidor.

El servidor se encarga de recibir y ejecutar los comandos que emite el cliente o enviarle información a este, como por ejemplo distancias captadas por un sensor.

Existen dos tipos de servidores en simulación, el servidor continuo, en el cual no se puede controlar su inicio y su final y el servidor temporal el cual se va a utilizar en este trabajo.

El servidor temporal se inicia desde un script de V-REP mediante la función "simExtRemoteApiStart" y finaliza o bien al acabar la simulación o mediante la función "simExtRemote-ApiStop"; es muy utilizado ya que el usuario es el que posee el control sobre el servidor de activarlo o desactivarlo.

4.6.2 Cliente.

El cliente se encarga de enviar los comandos al servidor en V-REP desde el programa externo, MATLAB, y, en algunos casos, recibir las respuestas de estos.

Este proyecto se ha centrado en la API remota de MATLAB, por lo que se va a explicar si implementación para este caso.

Para ello es necesario tener en nuestro directorio de trabajo los siguientes ficheros:

- remoteApiProto.m
- remAPI.m
- remotaApi.dll

Estos archivos se pueden encontrar en el directorio de instalación de V-REP siguiendo la siguiente ruta Programming/remoteApiBindings/Matlab separados según el tipo de procesador del sistema, es crucial instalar el mismo tipo de archivo que Matlab se use es decir 32 o 64 bits.

El procedimiento que seguir para crear la comunicación entre ambas plataformas se explica en el apartado 6.3 de este trabajo.

Las funciones desde MATLAB se utilizan prácticamente igual que en el lenguaje LUA la diferencia principal es la necesidad de concretar dos parámetros esenciales:

- **Modo de operación.** Indica la forma de ejecutar el comando en el simulador; todos los métodos de comunicación serán explicados en el siguiente apartado.

- **ClientID.** Este parámetro se obtiene mediante la llamada a la función "simxStart" y es necesario para todas las llamadas de funciones.

4.6.3 Modos de operación.

Mediante los siguientes modos de operación se puede saber si la sentencia ejecutada se ha realizado de forma correcta en V-REP. Los usos incorrectos de estos métodos pueden ralentizar el programa e incluso generar fallos.

Los distintos modos de operación son:

- **Non-blocking function calls**

Este método está diseñado para todas aquellas funciones que solo envíen datos a V-REP a través del servidor, sin necesidad de obtener una respuesta.

El modo de funcionamiento se explica en la siguiente imagen:

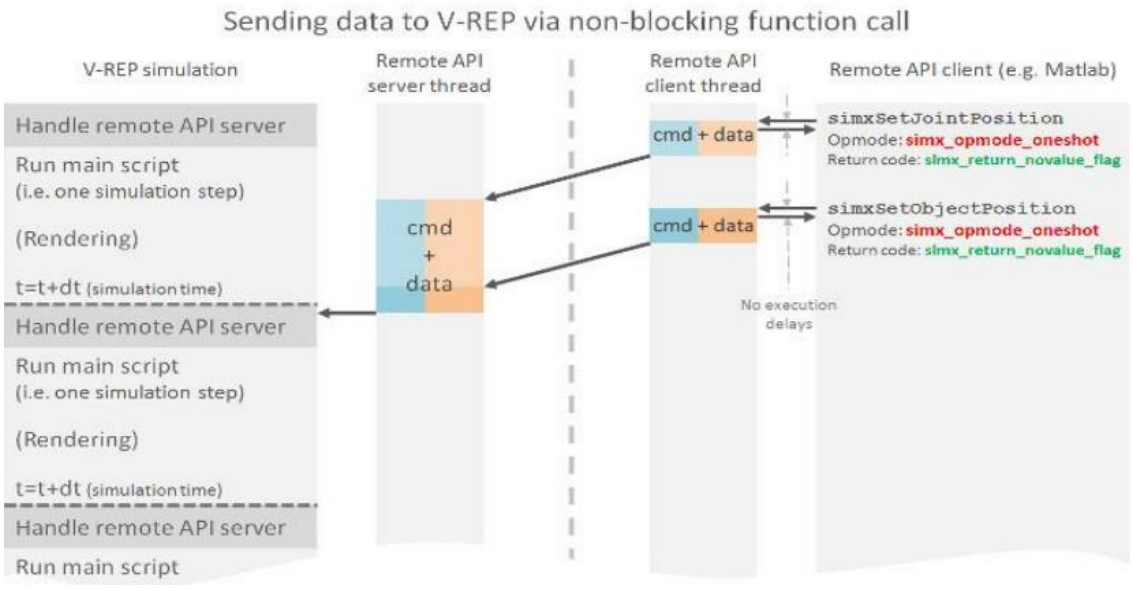

Ilustración 11, Non-blocking function calls [2]

- **Blocking function calls.**

Este modo está diseñado para todas aquellas funciones de la Api remota que necesitan de forma obligatoria una respuesta del servidor, como puede ser la recepción de una posición o de algún valor desde V-REP.

Su esquema de funcionamiento es el que sigue:

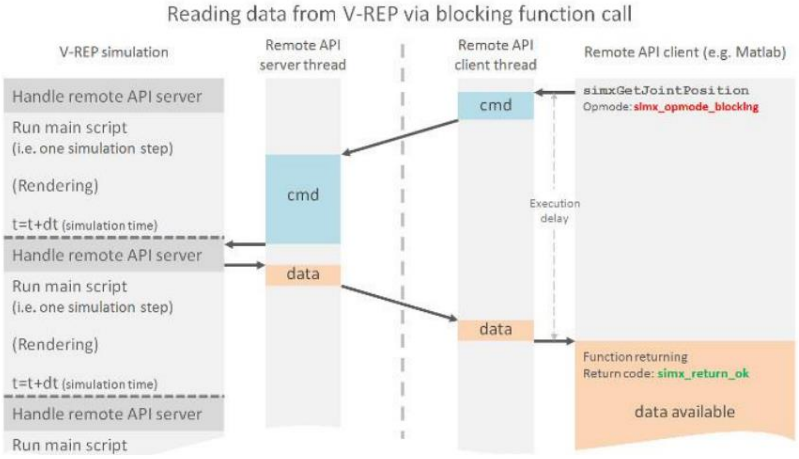

Ilustración 12, Ejecución blocking mode [2]

- **Data streaming.**

Esta creada para que el servidor sea capaz de anticipar las sentencias que el cliente requiere, de esta forma enviando después los datos por cada paso de simulación. Este método de transmisión es muy eficaz para una gran cantidad de datos sin producir retardos importantes en la conexión.

En la siguiente imagen se aprecia el modo de funcionamiento:

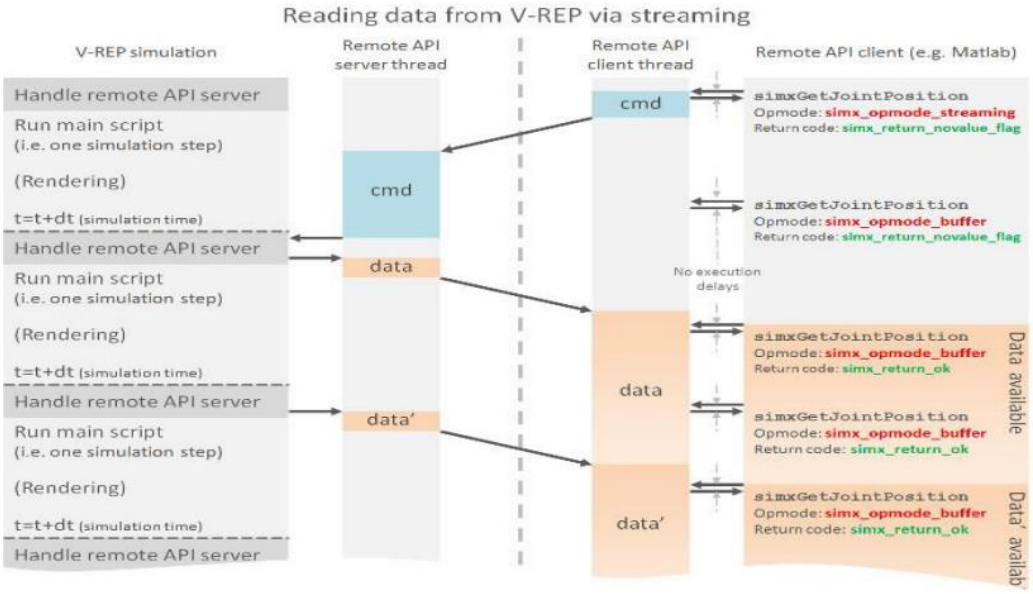

Ilustración 13, Ejecución mediante Data streaming [2]

- **Synchronous operations.**

Algunas veces la conexión entre servidor y cliente debe ejecutarse de forma sincronizada, para ello existe este modo de funcionamiento el cual nos aporta un control de la aplicación desde el cliente.

En el siguiente esquema se aprecia la forma de ejecutarse:

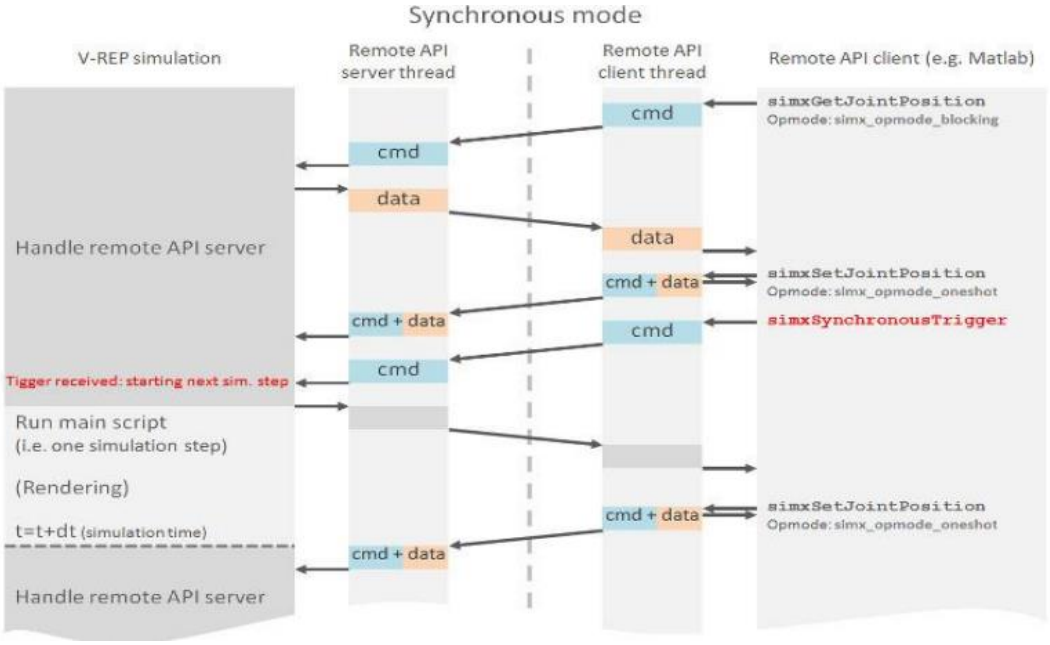

Ilustración 14,Synchronous mode [2]

5. Toolbox de Peter Corke

La Toolbox de robótica de Peter Corke es un instrumento increíblemente útil para el estudio de cualquier sistema robótico de no demasiado dificultad desde Matlab.

Esta herramienta es totalmente gratuita y muy fácil de instalar como se ha explicado en el apartado 6.3. Nos es suministrada directamente por su desarrollador a través de su página web en el apartado Toolbox. [5]

Esta toolbox incluye cerca de unas 400 funciones de todo tipo para el estudio de los robots, que además van actualizando, mejorando y añadiendo nuevas según se publican actualizaciones de la toolbox.

En el caso que nos ocupa hay que destacar las funciones enfocadas a la conexión entre Matlab y V-REP a través de la cual se puede manejar la simulación de forma remota incluyendo las opciones de envío y recepción de datos, ejecución de funciones, modificación de elementos, creación de nuevos elementos…

Debido a la imposibilidad de explicar en este trabajo todas las funciones las cuales nos pueden ser de utilidad, se ha añadió un anexo con todas ellas (Anexo 12: Funciones Peter Corke) para que cualquier interesado en este trabajo puede aplicarlas y seguir desarrollando cualquier tipo de aplicación.

Como se ha expuesto en este trabajo solo se entra a valorar las funciones que nos ayudan en nuestro trabajo con el entorno de simulación virtual V-REP, pudiendo encontrar el resto de las funciones de la Toolbox en el Libro de Peter Corke el cual facilita desde su página web. [5]

Aunque no es un tema que se toque en este trabajo, quiero mencionar la tremenda utilidad de las funciones de la biblioteca Serial Link ya que nos proporcionan un método de modelado tanto en 2D como en 3D de brazos robóticos o robots simples, pudiendo realizar la cinemática inversa o directa en ellos y realizarles un control muy preciso.

6. Proceso experimental.

En este apartado se va a describir el proceso seguido para el desarrollo de la parte práctica del proyecto, se explicará su realización de la misma forma que se procedió a su elaboración, es decir de forma escalonada.

La parte experimental del proyecto se subdivide en tres partes fundamentales:

- Modelado del brazo robótico IRB120 y de la herramienta.
- Control del brazo robótico IRB120 y de la herramienta InMoovSR.
- Conexión de la herramienta al brazo robótico.

6.1 Modelado del brazo robótico IRB120.

En primer lugar, se va a proceder a explicar el método de importación de archivos en 3D ya que este proceso es necesario para poder crear ambos modelos en el entorno virtual de simulación.

En el caso del brazo robótico, para comenzar el trabajo, obtuvimos el archivo URDF correspondiente al modelo en 3D del mismo. [4]

Como ya se explicó en la descripción del simulador V-REP, este nos ofrece la posibilidad de importar archivos para la generación de los modelos, en el caso que se estudia se ha decidido realizar dicha importación mediante los archivos URDF debido a la facilidad de uso de los mismos y a la fácil importación.

Para ello se accede dentro del menú principal del programa a la opción Plugins>>URDFimport.

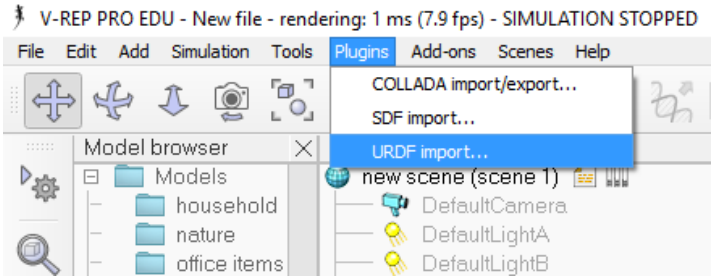

Ilustración 15,Importacion URDF

Al pinchar en esta opción se abre una ventana con varias opciones las cuales no deberán ser modificadas a no ser que se nos produzca un error en la generación del modelo en cuyo caso activaremos *"Show convex decomposition dialog"* donde se nos darán parámetros a modificar para la correcta importación.

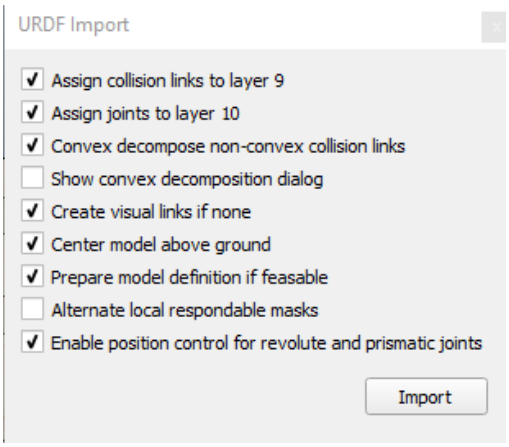

Ilustración 16,Dialogo URDF

En el caso que nos ocupa la importación es inmediata mediante el botón de importar que aparece en la ventana anterior.

Una vez importado hay que tener en cuenta que todas las articulaciones y las piezas del robot deben estar orientadas respecto al mismo sistema de referencia, ya que de esta forma podremos gestionar los movimientos de cada articulación de la forma adecuada.

Para ello seleccionamos la opción de reorientar grupo respecto al sistema de referencia de "mundo", como se ve en la imagen siguiente.

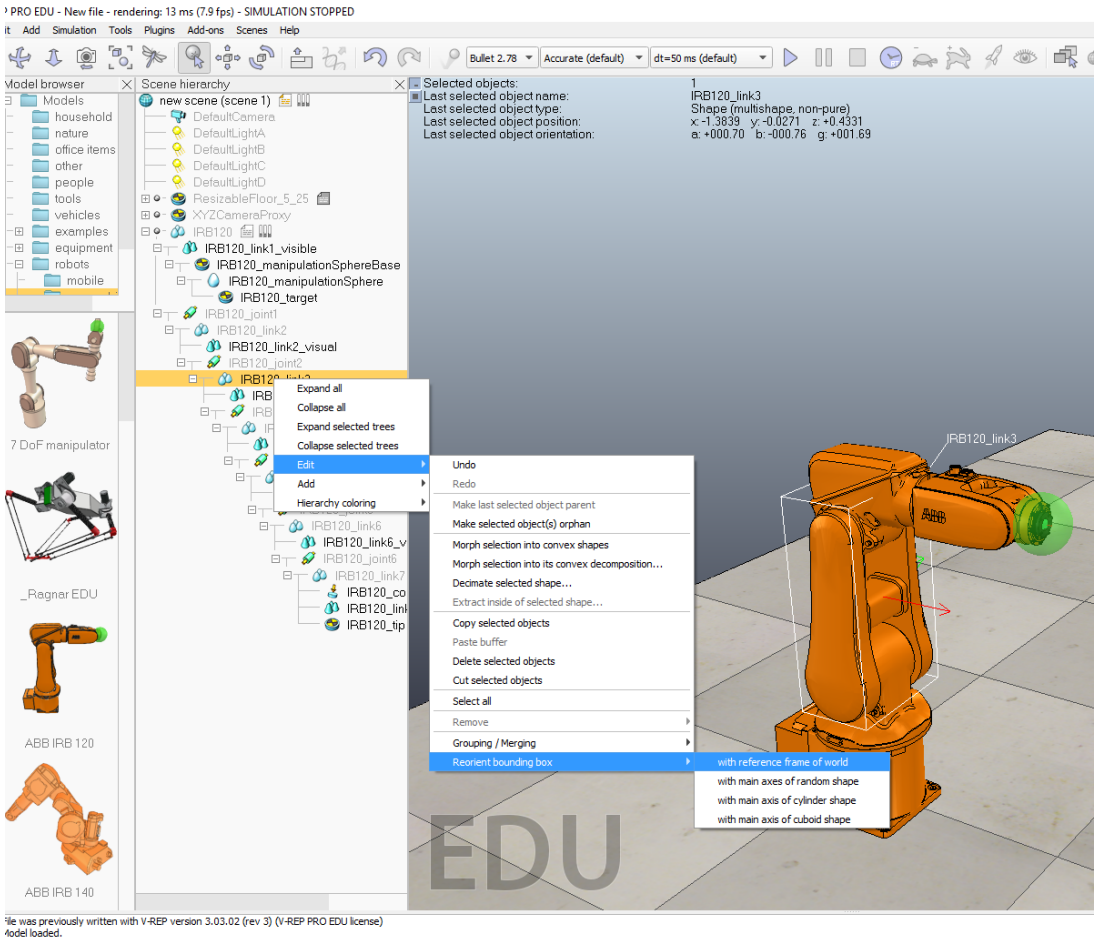

Ilustración 17,Orientar objetos

Cabe destacar que este proceso hay que realizarlo tanto con las articulaciones como con los links (eslabones) del robot.

Esta parte es fundamental y hay que tener en cuenta que también se deberá ejecutar este proceso a la hora de modelar la herramienta.

Es preciso mencionar que, si al importar mediante el URDF el brazo robótico no se ha generado la jerarquía de la escena correcta, se debe organizar manualmente cada eslabón con su eje agrupándolo de forma genérica en un grupo que finalmente será el brazo robot.

En este caso además se han añadido cuatro nuevos objetos al brazo robótico los cuales funcionaran como referencia del efector final, ya que se piensa que puede ser útil en futuras aplicaciones en las cuales mediante un código especifico se puede programar los movimientos del robot para una posición determinada de dicho efector. Esta modificación consta de 3 objetos dummy de los cuales dos funcionan creando la conexión entre objetivo y el robot mediante cinemática inversa (IRB120_tip e IRB120_target), además se ha añadido otro dummy como grupo de objetos para ligar el objetivo a una referencia visual en forma de esfera.

En la siguiente figura se observa la configuración que deben tener los dos dummy objetivo y objeto:

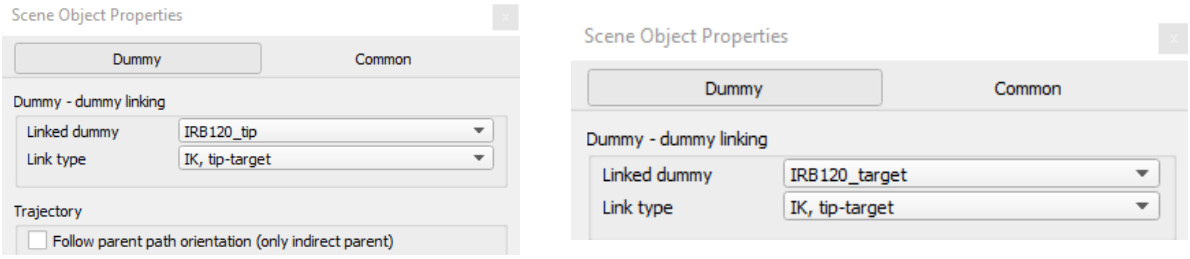

Ilustración 18, Configuración de dummy

De esta forma la jerarquía de la escena quedara de la siguiente forma:

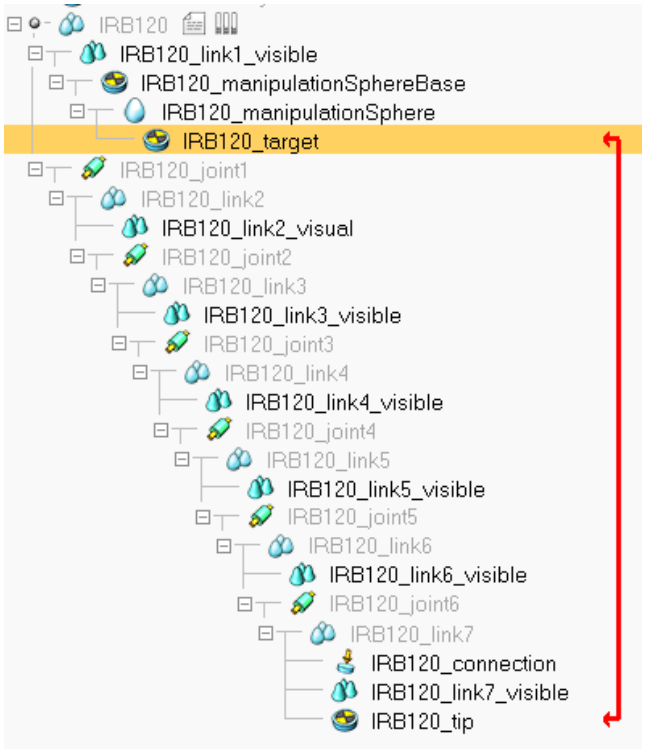

Ilustración 19,Jerarquía modelado IRB120

Una vez estructurado el robot correctamente, surgen los primeros problemas a la hora de simular, esto es debido a que realmente el robot esta creado visualmente, pero sus partes no tienen las características adecuadas para simular un robot físicamente.

En este caso se apreció como el robot, al iniciar la simulación, se movía del lugar donde se había colocado atravesando el suelo y hacia a delante por la inercia generada por el propio brazo, llegando así a volcar y rodar hasta salirse de la escena.

De esta forma se percibe la mala configuración de los objetos que formaban nuestro brazo robótico, y mediante la comparación con otros modelos suministrados por Coppelia (IRB140, dado su razonable parecido con nuestro robot) localizamos los errores en la configuración y se subsanaron dichos errores. De esta forma las diferentes partes de nuestro robot quedan configuradas de la siguiente forma:

- Las articulaciones se configuran en el modo de cinemática inversa con modo hibrido de operación, funcionando mediante un control PID.
  En el caso de las dos articulaciones circulares completas "joint1" y "joint6" se selecciona la opción de operación cíclica.

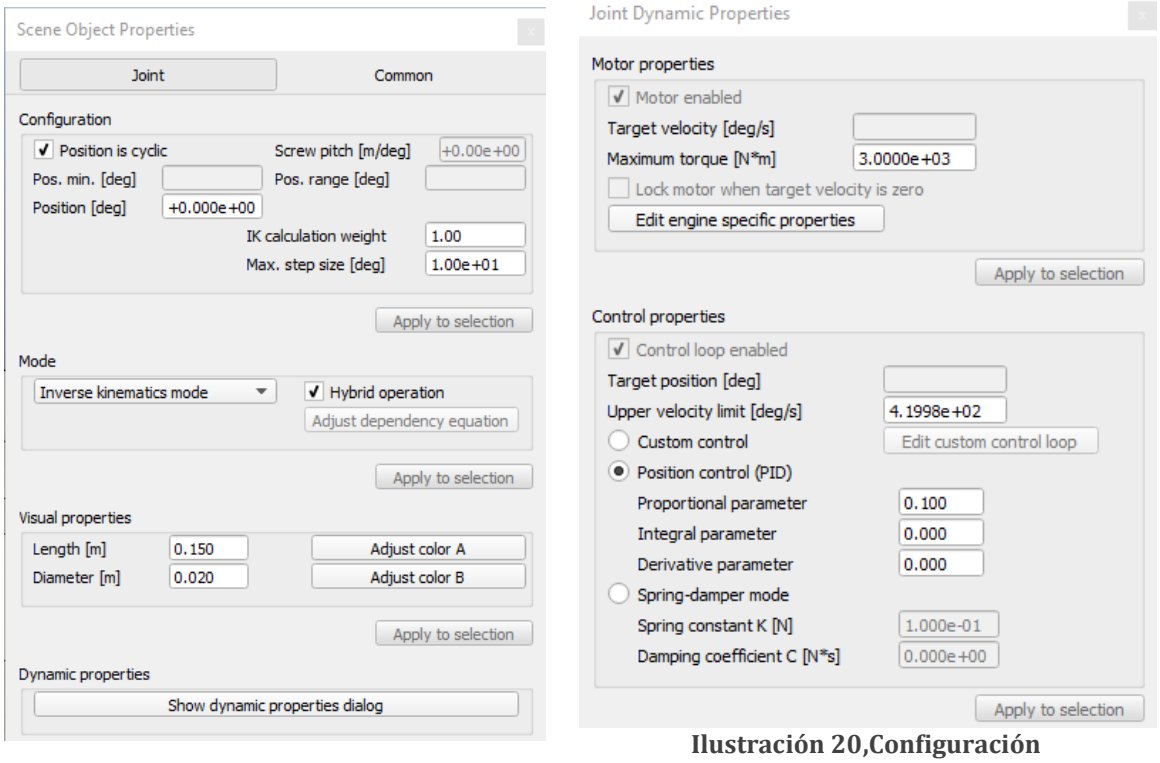

Ilustración 20,Configuración articulaciones cíclicas

- En lo correspondiente a los eslabones del robot, como ya hemos visto en la jerarquía, se dividen en dos partes, la parte visual y con efectos de medida, y la parte estructural de organización y jerarquización del programa.
  La parte de organización se configura para que se comporte de forma dinámica y con la capacidad de responder ante una acción externa, pero no se activara en este caso ninguna opción especial de los objetos. Por tanto, su configuración queda de la siguiente forma:

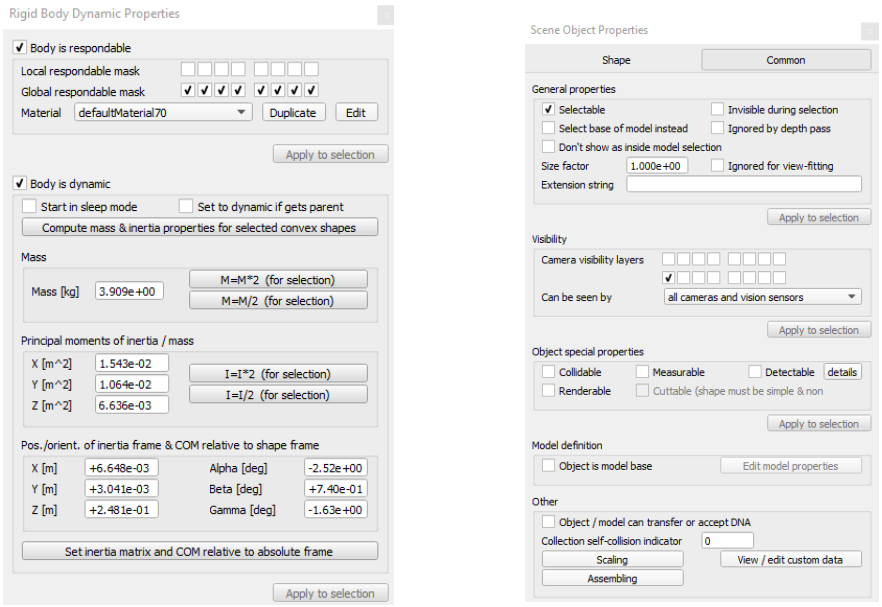

Ilustración 21,Configuración eslabones principales

En el caso de los eslabones que forman la parte visual del robot no se les añade las características dinámicas de los anteriores, pero en su caso si que se les fija las propiedades especiales para que estos se comporten de tal manera que puedan colisionar entre ellos advirtiéndonos del suceso se puedan detectar o medir cualidades físicas:

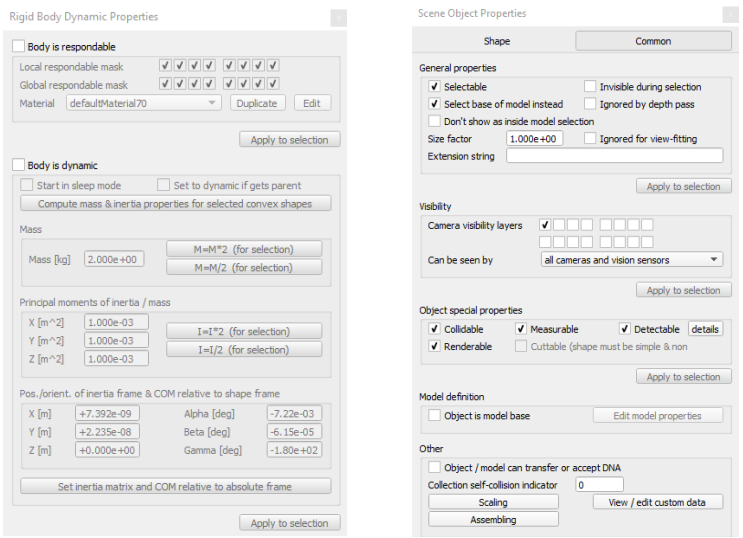

Ilustración 22, Configuración eslabones visuales

Una vez seguido todo este proceso el modelo ya está preparado para ser guardado y para su futuro uso en simulaciones de situaciones reales, en el caso que se trata queda de esta forma correctamente configurado para su posterior manejo desde el software MATLAB mediante la conexión API remota que será explicada más adelante.

Para poder disponer de este modelo en el explorador de modelos que nos ofrece V-REP se procederá como sigue:

- Seleccionamos el robot en la jerarquía escogiendo el objeto nombrado IRB120 que engloba todo nuestro modelo, a continuación, en el menú principal>>file>>sabe_model_as… guardamos el modelo con la imagen escogida en la dirección correspondiente a los modelos de V-REP:

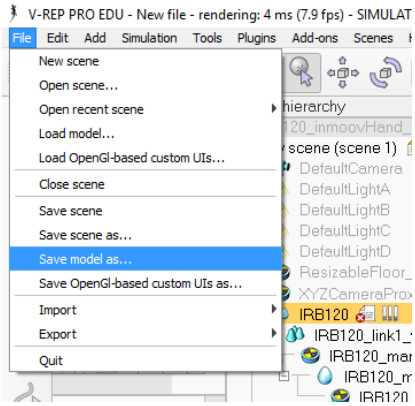

Ilustración 23, Guardado de modelo

Se abrirá el siguiente cuadro de dialogo para escoger la imagen deseada del robot:

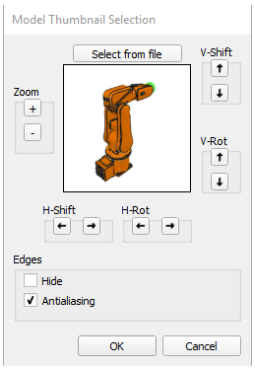

Ilustración 24, Icono del modelo

Con esto ya tenemos el modelo correspondiente al IRB120 diseñado y configurado para su fututo uso.

6.2 Modelado de la herramienta Inmoov-SR

El modelado de la herramienta es muy similar al del brazo robótico, ya que gracias a los datos facilitados en diversas páginas de internet [5] [6] con información muy precisa de su montaje, así como la suministración de los archivos en 3D que conforman la mano.

Hay que mencionar que se nos ha facilitado el modelo 3D de la mano, del soporte modificado para una mejor adaptación al brazo robótico y de los tornillos por medio de dos alumnos los cuales llevaron a cabo la impresión y montaje de la mano en su trabajo de fin de grado. [7]

Una vez generados los ficheros URDF (INMOOV-SR.urdf adjunto en archivos digitales) de los objetos citados, se procede a la importación de los mismos siguiendo el mismo método descrito en el apartado anterior para el modelado del IRB120. También es posible su generación mediante la importación de los modelos en 3D desde el meno archivo importar objeto.

Por lo tanto, la escena que tenemos es la siguiente:

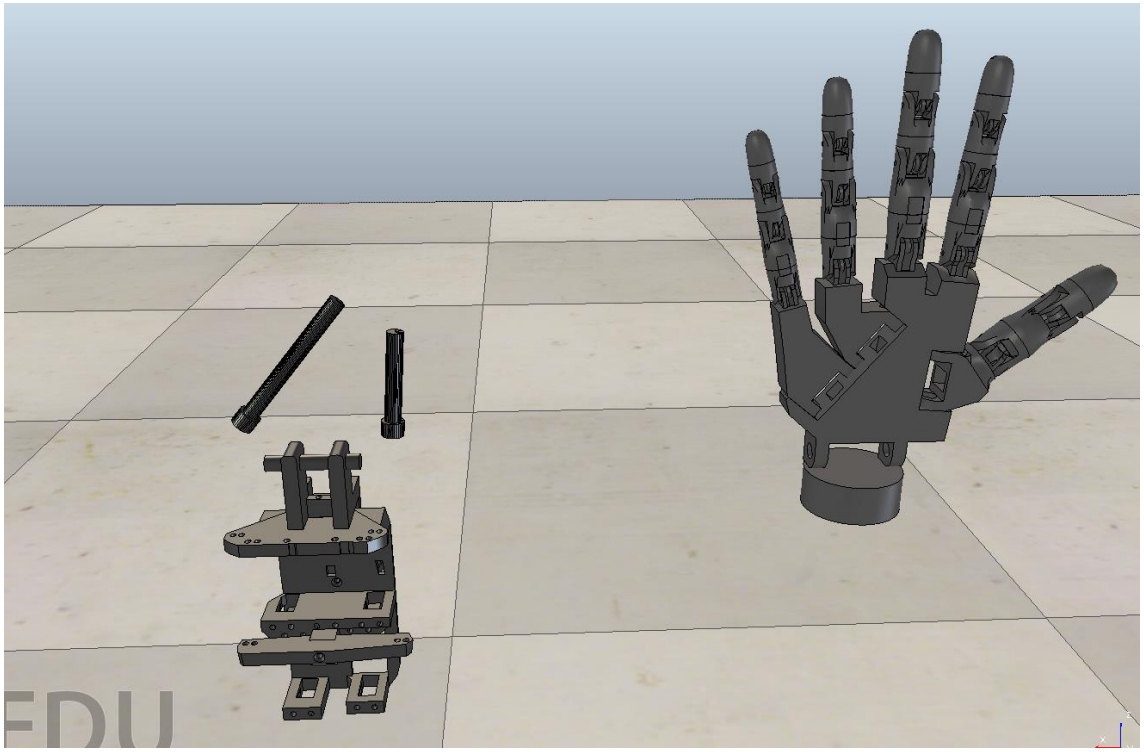

Ilustración 25, Objetos importados para creación de Inmoov-SR

Como se puede observar es necesario la reorganización de los elementos de tal forma que se elimine la actual muñeca de la mano y se le adapte la nueva muñeca con un soporte que nos ofrece mayor variedad de posibilidades de uso una vez conectado al brazo robótico. Es necesario también la modificación de la posición de los tornillos, que, aunque en el simulador su función es meramente estética hemos querido realizar el modelado l más preciso posible.

Para realizar estas pequeñas modificaciones se procede una forma muy simple ya que el simulador nos ofrece una herramienta específica para modificar la posición y la orientación de los objetos.

Para modificar la posición de un objeto se nos ofrecen varias posibilidades como se ve en la imagen inferior, podemos desplazarnos directamente a un punto respecto a una referencia fija ya sea la global (world) o la de una pieza (frame), y además se ofrece la posibilidad de desplazarte poco a poco una distancia predeterminada a lo largo de los ejes de ambos sistemas de referencia. Esta última opción es muy útil a la hora de precisar correctamente la posición de los tornillos que conforman la mano y la muñeca, ya que de no ser preciso el ajuste el programa puede detectar una colisión entra las piezas.

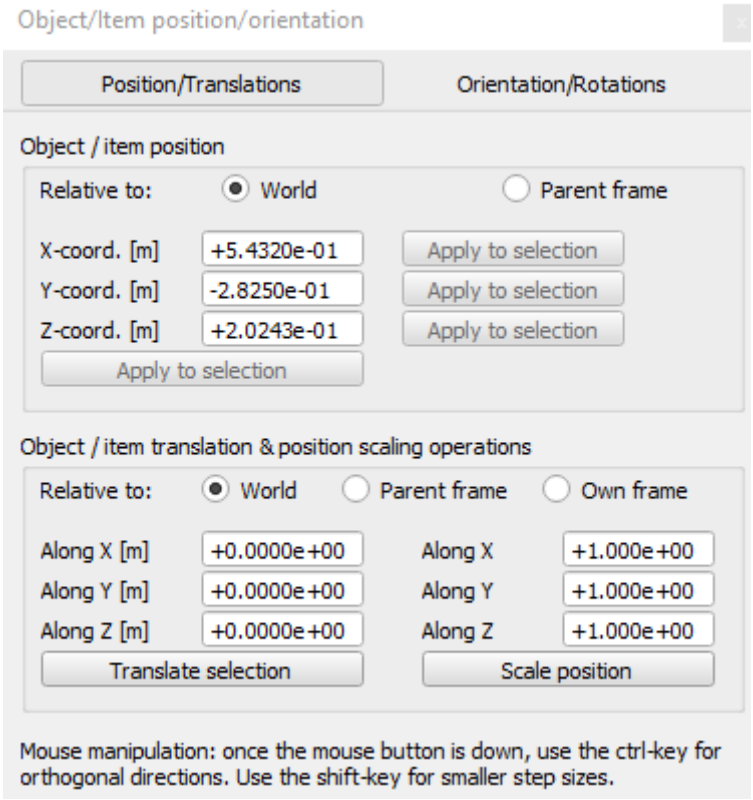

Ilustración 26, Ventana de modificación de objetos

Una vez realizado este paso la mano se encuentra visualmente montada quedando idéntica a la herramienta de la cual disponemos en el laboratorio, pero como se explicó en el caso anterior no solo hay que atender a los aspectos visuales ya que la jerarquía de la escena no está ordenada de la forma correcta, esto se puede observar al desplazar la mano ya que si no reorganizamos la jerarquía los objetos nuevos que hemos situados se quedaran en el mismo sitio haciendo inútil el trabajo anterior.

De esta forma es importante situar las piezas de la forma que queremos que respondan después en la simulación. En este caso es sencillo ya que son piezas que no van a realizar actividad en la simulación más allá de la estructural, y que además son la base del modelo ya que las articulaciones móviles correspondientes a los dedos parten todas de estos puntos.

Por tanto, la jerarquía de la escena correspondiente a la herramienta Inmoov-SR quedaría con la siguiente estructura:

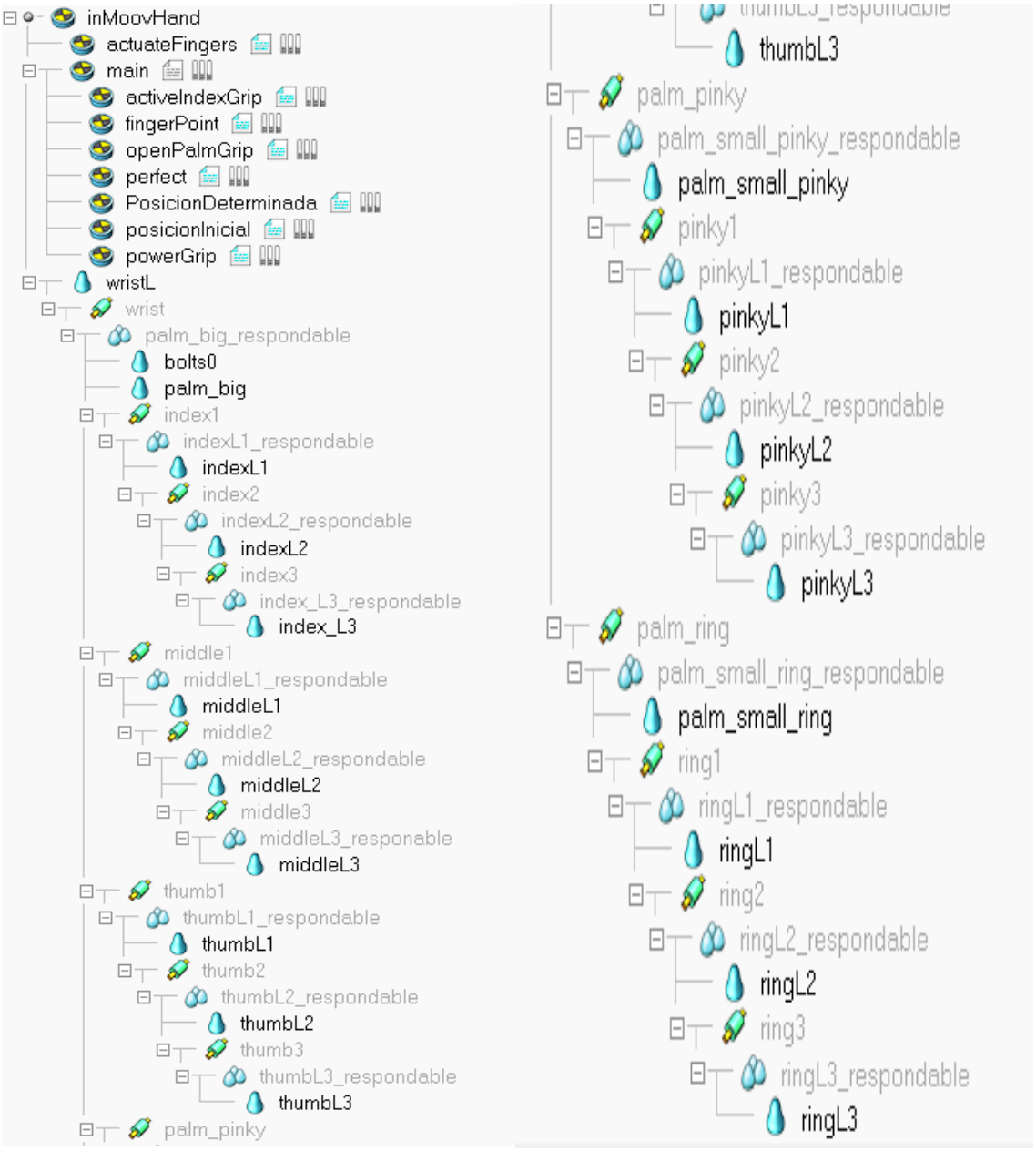

Ilustración 27, Jerarquía de la herramienta

Se puede observar que se han añadido una serie de dummy los cuales no tienen funcionalidad en el modelado ya que han sido utilizados como objetos auxiliares en el control de la mano asignándoles a cada uno de ellos una acción para la ejecución por parte de la herramienta, así como uno principal que engloba las acciones y uno encargado de la gestión de la interfaz personalizada. Este tema se tratará en el apartado designado al control de la herramienta.

Como en el caso del robot se han configurado los ejes de cada elemento móvil con respecto al sistema de referencia global, según se explicó en el apartado correspondiente del modelado del brazo IRB120.

Con respecto a la configuración de cada elemento de la mano se ha procedido a los ajustes de las propiedades de la siguiente forma:

- Todas las articulaciones tienen la misma configuración, en este caso funcionan en el modo de torque/fuerza, con un control de velocidad mediante una regulación PID. Al ser las articulaciones rotacionales y con un movimiento singular en dos sentidos se han configurado como cíclicas.
  Con todos estos parámetros las ventanas de configuración quedan de la siguiente manera:

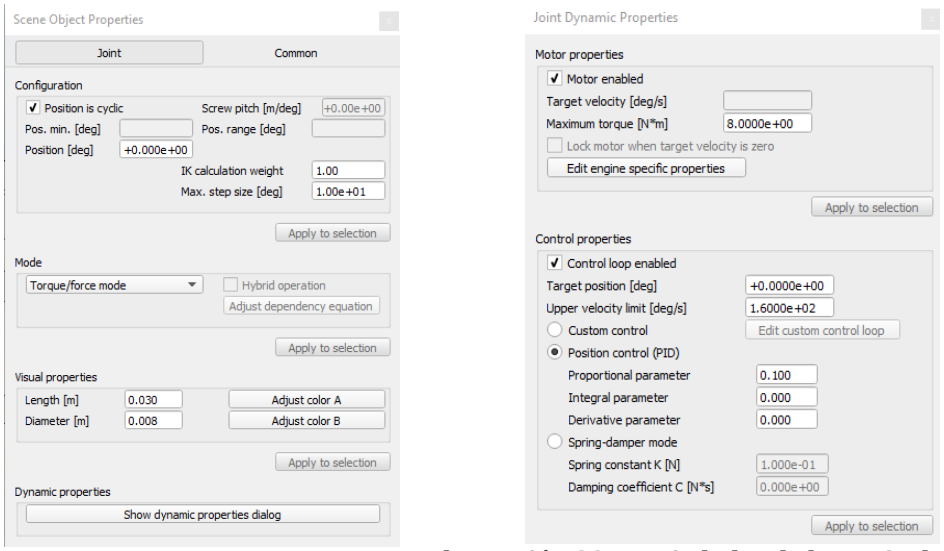

Ilustración 28,Propiedades de las articulaciones de la herramienta

- Con la parte que corresponde a los eslabones se observa un caso muy similar al del IRB120, ya que, cada eslabón, consta de una parte con propiedades dinámicas de la cual "cuelga" en la jerarquización el resto de eslabones de cada dedo pero que en este caso también tiene las propiedades especiales de los objetos y otros objetos los cuales solo tienen las propiedades especiales del simulador que actúa como referencia visual, es decir es la parte que el simulador utiliza para el cálculo de posiciones, la existencia o no de colisiones …

  Por tanto, los eslabones dinámicos quedan con las siguientes propiedades:

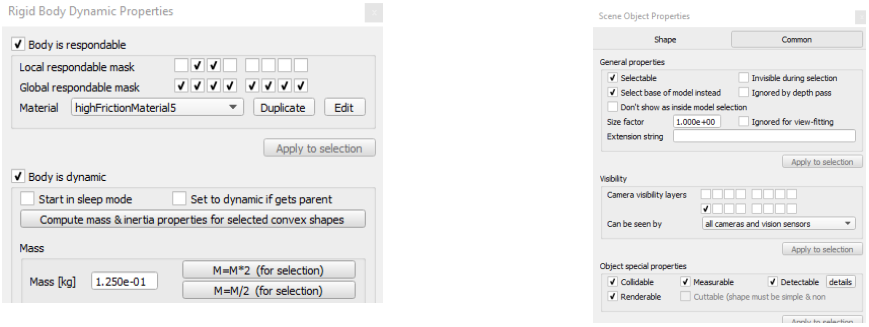

Ilustración 29, Configuración dinámica eslabones

Mientras que de formas de los eslabones se mantienen de la siguiente forma:

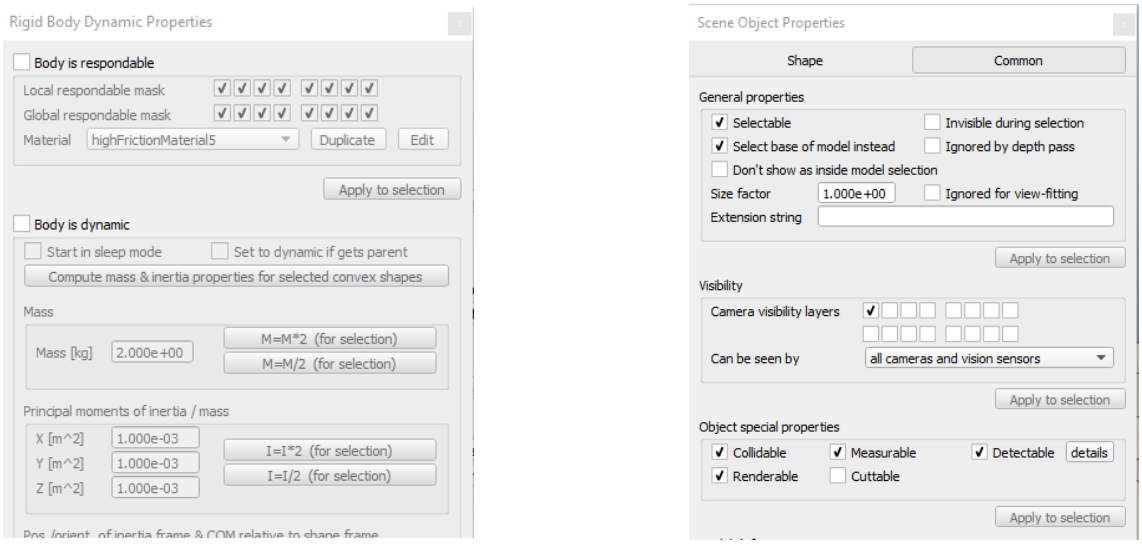

Ilustración 30,Configuración eslabones

De esta forma, a falta de explicar en el control de la mano los dummys introducidos en el modelado de la misma, quedaría perfectamente modelada para su uso tanto como una herramienta auxiliar de un brazo robótico como para la simulación de movimientos como un elemento independiente.

Una vez que la mano esta modelada únicamente queda realizar el proceso de guardado del modelo para que sea posible usarlo todas las veces que nos sea necesario. Para ello se seguirán los pasos explicados en el apartado anterior cuando se explicó dicho método para el IRB120.

En este proyecto se ha generado el modelo de la mano con el nombre Inmoov-SR2.4.ttm que es posible encontrar en los archivos digitales adjuntos.

6.3 Control del brazo robótico IRB120.

Para el control del brazo robótico se ha optado por realizar un control únicamente remoto, es decir, vamos a programar los movimientos del brazo desde un software externo al simulador virtual, en nuestro caso desde MATLAB.

En primer lugar, hay que realizar un proceso sencillo el cual nos permite interactuar entre ambas plataformas.

Para la conexión entre V-REP y MATLAB vamos a necesitar dos recursos disponibles en estas plataformas, el principal encargado de la conexión entre ellas es la función remoteAPI de V-REP y en el caso de MATLAB será necesaria la Toolbox de robótica creada por Peter Corke [8]

Todos los archivos necesarios para esta simulación se encuentran en la siguiente ruta: la escena para la simulación en archivos digitales>>ESCENAS V_REP>>IRB120; los archivos de MATLAB necesarios en archivos digitales>>ARCHIVOS MATLAB

La instalación de la Toolbox en MATLAB sigue un procedimiento muy simple únicamente hay que:

- Descargar de la página mencionada en la bibliografía [8] el archivo con la Toolbox
- Descomprimir dicho archivo.
- Copiar el archivo en la siguiente carpeta: C:\Program Files\MATLAB\R2016b\toolbox
- Crear un script en MATLAB el cual inicialice el funcionamiento de la Toolbox, en nuestro caso dicho código es el siguiente, siendo necesario llamar a la función cuando iniciemos MATLAB:

ini_robotics.m

```
1 -    addpath ('C:\Program Files\MATLAB\R2016b\toolbox\rvctools');
2 -    startup_rvc
```

Ilustración 31, Instalación Toolbox

- De esta forma queda correctamente instalado la Toolbox para su uso en MATLAB, pero para que se puede realizar la sincronización entre ambos softwares es preciso modificar la función VREP.m que nos ofrece Peter Corke siendo preciso añadir el comando "`opt.path='C:\Program Files (x86)\V-REP3\V-REP_PRO_EDU'`" en la línea 125 de dicho script.

Una vez realizado este proceso ya solo nos falta configurar V-REP para realizar la conexión, este caso es mucho más simple ya que basta con añadir el código:

```
13 -- Initialization part (executed just once, at simulation start) ---------
14 if (sim_call_type==sim_mainscriptcall_initialization) then
15     simOpenModule(sim_handle_all)
16     simExtRemoteApiStart(19999)
17     simHandleGraph(sim_handle_all_except_explicit,0)
18 end
19 ------------------------------------------------------------------------
```

"simExtRemoteApiStart(19999)" en el MainScript de la escena en cuestión, modificándolo en la parte de iniciación, quedando la misma como sigue:

Ilustración 32,Configuración API remota

Como el control que principalmente ocupa al trabajo es el de herramienta Inmoov-SR, se ha decidido realizar el control del IRB120 mediante el envío y recepción de los datos correspondientes a las articulaciones, es decir podremos trasmitirle cual queremos que sea el ángulo de cada articulación, así como saber con que ángulo se encuentran en cada momento.

Este método de control es de gran utilidad ya que mediante los métodos cinemáticos podremos según la configuración de las articulaciones "q" obtener la posición del robot. Es posible también a la inversa, si necesitamos obtener la configuración de las articulaciones para colocar el efector final del robot en un punto determinado, bastaría con utilizar un modelo del robot creado en MATLAB y el método de la cinemática directa para determinar los parámetros que le pasaremos al robot para que se situé en dicha posición.

El modelado y uso de los métodos cinemáticos tanto directa como inversa está desarrollado en las prácticas de laboratorio de la asignatura Sistemas Robotizados impartida para el Grado de Ingeniería Electrónica y Automática Industrial de la Universidad de Alcalá, así como en un trabajo de fin de grado de la misma facultad. [9]

Se adjunta en formato digital los archivos correspondientes al modelado del brazo robótico IRB120 y su control mediante cinemática inversa y directa.

Un ejemplo de código en MATLAB para el control remoto del IRB120 es el siguiente:

```
                % CONEXION MATLAB - V-REP %
 % ENVIO Y RECEPCION DE LA CONFIGURACION DE LAS ARTICULACIONES %
%-----------------------------------------------------------------%

%Se inicia la ToolBox de Peter Corke
ini_robotics

%conexión con el entorno de simulación
vrep=remApi('remoteApi')
v=VREP()

%obtención del manejador del robot y configuración como brazo robótico

robot=v.gethandle('IRB120')
arm=v.arm('IRB120')

%obtener array de configuraciones actuales del robot

configuración=arm.getq()

%transmisión de nueva configuración al robot
```

```
nueva_config= [0 90 0 45 45 90]
arm.setq(nueva_config)
```

Se va a explicar de forma reducida las funciones utilizadas en el código de ejemplo y alguna otra que puede ser de interés para el control de un brazo robótico, para más información sobre todas las funciones que nos ofrece Peter Corke ver el Anexo 12: Funciones Peter Corke.

Las principales funciones utilizadas son las siguientes:

- **V=VREP( )**

Es la función encargada de crear la conexión con el simulador de V-REP, dicha conexión la realiza en función a unos parámetros fijados en él .m, como es el path mencionado anteriormente que es necesario incluir en el fichero.

A parte de la ruta hasta el programa V-REP necesita de otros 4 parámetros fijados por defecto:

  - Timeout fijado en 2000ms, es el tiempo de espera sin respuesta, al agotarse dicho tiempo se genera un error en la conexión.
  - Cycle, por defecto 5ms, es el periodo entre actualización de datos entre ambas plataformas.
  - Port, puerto de enlace entre los programas debe estar configurado el mismo en los dos softwares
  - Reconnect, se puede activar o desactivar una reconexión automática en caso de error.

- **V.arm('name')**

Este método nos reconoce como un brazo robótico a un robot dentro de la simulación con el nombre introducido en el parámetro. Gracias a este método podremos usar una gran variedad de funciones las cuales nos facilitaran el control del robot.

- **V.getHandler('name')**

Devuelve el manejador interno del robot "name" de V-REP esta opción es necesaria debido a la necesidad de usar los manejadores de algunas funciones en lugar del nombre fijado por el usuario.

- **X=arm.getq( )**

Obtiene los valores de las articulaciones del brazo robot fijado en el método arm guardándolos en forma de array de ángulos con tantas posiciones como articulaciones tiene el brazo robótico.

- **Arm.setq(x)**

Fija la configuración de las articulaciones del brazo robot fijado en el método arm a unos valores pasados en forma de array de ángulos con tantas posiciones como articulaciones tiene el brazo robótico.

- **Arm.teach(*options*)**

Esta función es muy útil ya que podemos modificar mediante un slider o modificando los valores de forma manual la configuración de las articulaciones mediante un panel generado por MATLAB de forma automática.

Las opciones que se pueden modificar de esta función son:

  - Las unidades que usar en la configuración de las articulaciones, por defecto viene dado en radianes, pero bastaría con escribir 'degrees' como opción para manejar las articulaciones en grados.
  - La segunda opción que podemos manejar es introducir una posición inicial mediante un array.

Un ejemplo de la visualización de esta función es el siguiente:

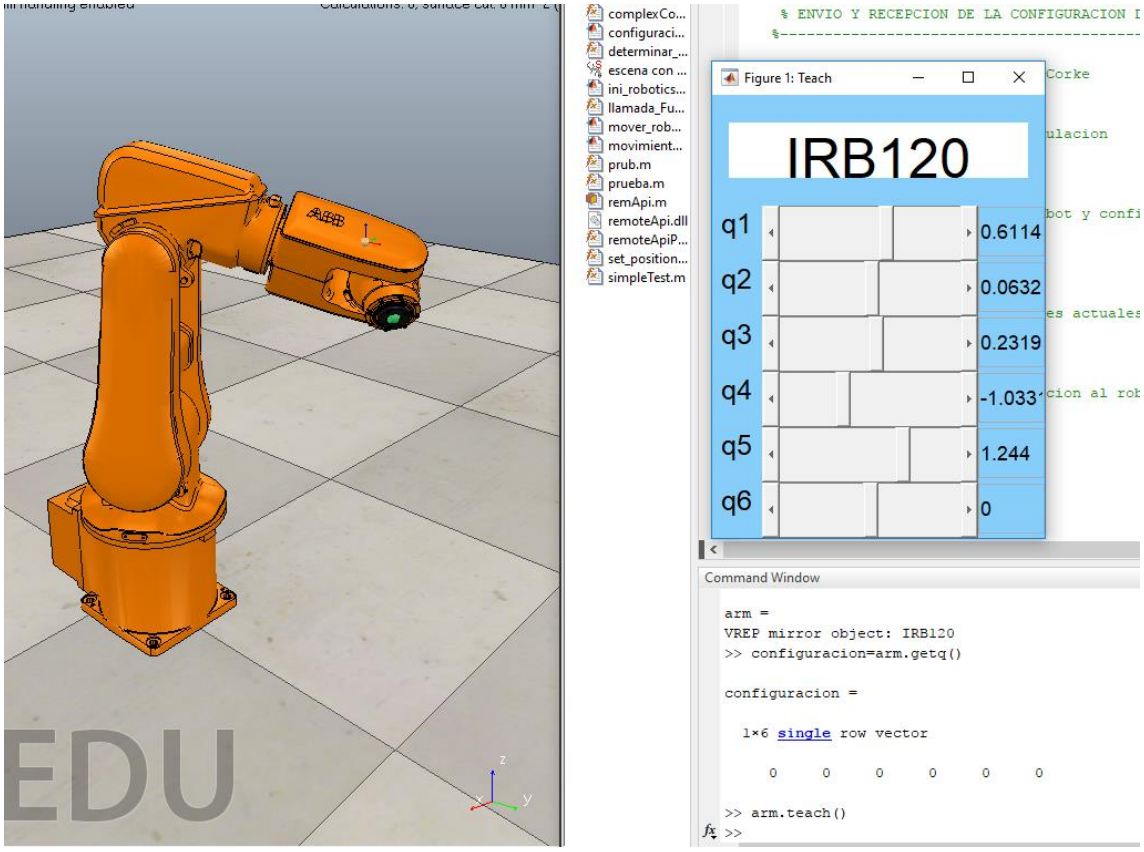

Ilustración 33,Control remoto IRB120

6.4Control de la herramienta Inmoov-SR.

Por último, queda explicar el control realizado a la herramienta.

Todos los archivos necesarios para esta simulación se encuentran en la siguiente ruta: la escena para la simulación en archivos digitales>>ESCENAS V_REP>>Inmoov-SR_control.ttt; los archivos de MATLAB necesarios en archivos digitales>>ARCHIVOS MATLAB

Se ha optado por realizar un control mediante scripts internos de V-REP, pero añadiendo la posibilidad de modificar desde MATLAB los parámetros necesarios para ajustar la posición de cada dedo.

Se ha diseñado por otra parte una interfaz personalizada desde la cual podemos escoger la posición que queremos que tome la mano dentro de las diferentes posiciones por defecto que hemos creado, o mediante una serie de sliders modificar cada una de las articulaciones por separado o en grupos de cada dedo.

Se va a comenzar explicando esta última interfaz gráfica, desde su diseño hasta la escritura del código necesario para leer los datos.

6.4.1 Control mediante interfaz personalizada.

El control mediante una UI nos permite la variación de los parámetros de las articulaciones a través del ajuste de unas sliders vinculados a cada una de ellas.

A demás, se ha creído conveniente la inserción de uno slider adicional en cada dedo el cual simula el movimiento real de la mano, transmitiendo a cada articulación del mismo dedo la misma configuración, esto se debe a que nuestro modelo real no consta de un control independiente por articulación ya que funciona con un servo por dedo el cual mediante un hilo de nailon mueve cada dedo.

Se han añadido como opción extra unos botones en la parte derecha de la interfaz los cuales activaran uno de los movimientos predefinidos que vamos a incluir en la mano mediante los scripts del modelo.

En primer lugar, se va a proceder a explicar el método por el cual se ha creado una interfaz gráfica personalizada para el control de la mano.

Se accede al menú de los UI mediante el siguiente icono situado en la barra de herramientas:

Ilustración 34,Icono UI

Al hacer clic en él se nos abre la pantalla de configuraciones en la cual en este momento no se tendrá ningún UI creado por el usuario, solo nos aparecerá el creado por defecto para el ajuste del tamaño del suelo:

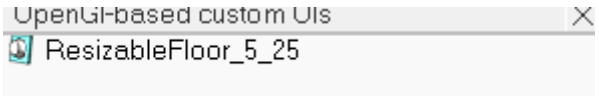

Ilustración 35,UI definido por el programa

Para la creación de un nuevo UI pinchamos en la opción “Add new user interface” la cual aparece en la siguiente ventana de forma automática:

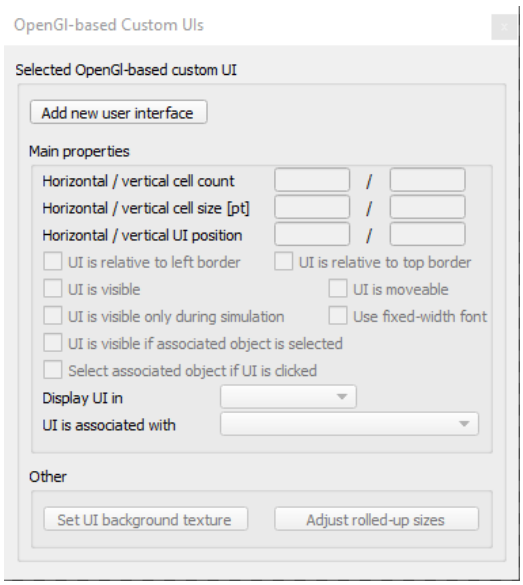

Ilustración 36, Crear nuevo UI

A continuación, se insertarán los valores que definen el tamaño de nuestro interfaz gráfico, así como una selección de diferentes opciones eligiendo las más apropiadas para nuestro caso.

En este proyecto se ha decidido que el interfaz grafico conste de botón de cerrado y de minimización, por lo que dejaremos señaladas estas opciones en el dialogo que se nos abre.

Con respecto al tamaño del UI se ha elegido uno intermedio seleccionando una altura y anchura de celda de 8 (pt) estando el interfaz constituido por 50 celdas horizontales y 51 celdas verticales. La posición del interfaz va a ser un poco indiferente ya que fuera de la página de creación de la mismas la podremos mover libremente por la pantalla mediante la selección de la opción “UI is moveable”.

De las diferentes posibilidades que se nos dan se ha decidido seleccionar las siguientes:

- El interfaz grafico es visible únicamente en simulación.
- Al utilizar el UI se seleccionará el objeto al que pertenece.
- Es visible en todas las páginas.

Es necesario asociar el UI con la herramienta para que al programar el script correspondiente se puedan realizar los movimientos definidos en él.

Para finalizar con este cuadro de dialogo accederemos mediante el botón “Set UI background texture” a una nueva ventana en la cual cargaremos una imagen para que sea nuestro fondo de la interfaz gráfica. Esta imagen se adjunta en los archivos digitales como UIBackground.jpg

Por tanto, la ventana de configuración de la interfaz gráfica queda como sigue:

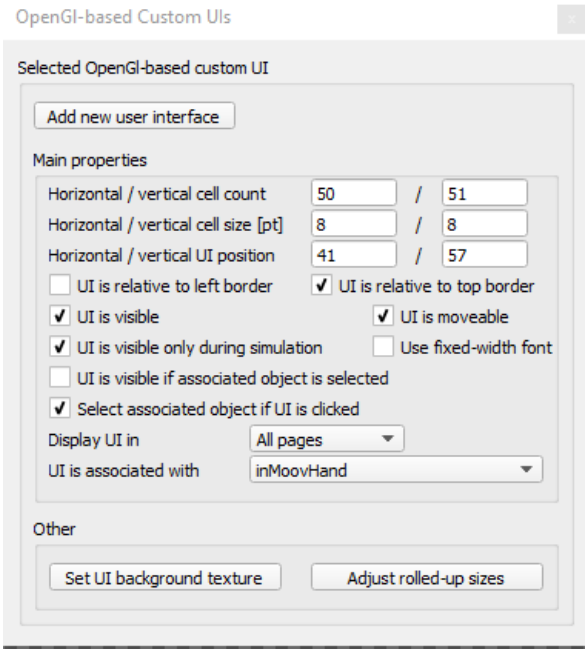

Ilustración 37, Configurar UI

Una vez llegado a este punto se van a generar los botones y las sliders a través de los cuales se va a controlar la simulación.

Se va a explicar el método de diseño de un botón y un slider ya que es el mismo proceso para cada uno de los elementos que se quieran tener:

En primer lugar, se seleccionan tantas celdas como sean necesarias para obtener el tamaño del botón deseado, para ello pulsamos las celdas mientras se mantiene pulsado CTRL, de esta forma se obtiene la forma del botón señalada en amarillo:

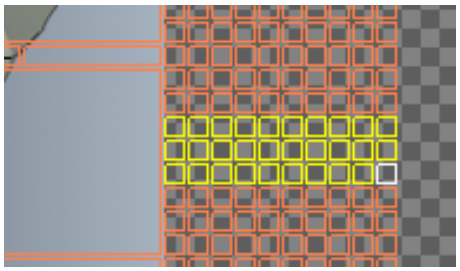

Ilustración 38,Crear botón en UI

Una vez seleccionadas las celdas a convertir, se pulsará al botón “Insert merged button” el cual se encuentra en una ventana auxiliar que se abre de forma automática al seleccionar las celdas.

De esta forma las celdas se unifican y se nos permite en el cuadro de dialogo modificar los distintos parámetros:

- Button handle: Nos permite identificar mediante un numero el botón creado, para que de esta forma podamos acceder a el a través de las diferentes funciones internas de V-REP para la programación de los movimientos en el script.

- Button type: En este apartado se nos da la opción de escoger el comportamiento del elemento creado (botón, slider, etiqueta o cuadro editable) en el caso que nos ocupa únicamente vamos a utilizar la opción de botón o slider, ya sea para la ejecución de los scripts predeterminados o bien para modificar manualmente la configuración de las articulaciones.

  Tanto en el caso de crear un botón como de crear un slider el proceso es similar solo sería necesario la modificación en el desplegable type para escoger el tipo d elemento.

  En lo correspondiente a las opciones que se pueden habilitar se mantienen las marcadas por defecto, ya que así queda bien implementado visualmente.

- Button Labels: Esta opción solo se va a usar en el caso que el elemento creado sea del tipo botón, ya que nos da la posibilidad de escribir un texto en cada uno de ellos.

Por tanto, con todo lo explicado anteriormente la configuración de un slider y de un botón debe ajustarse a la recogida en las siguientes imágenes:

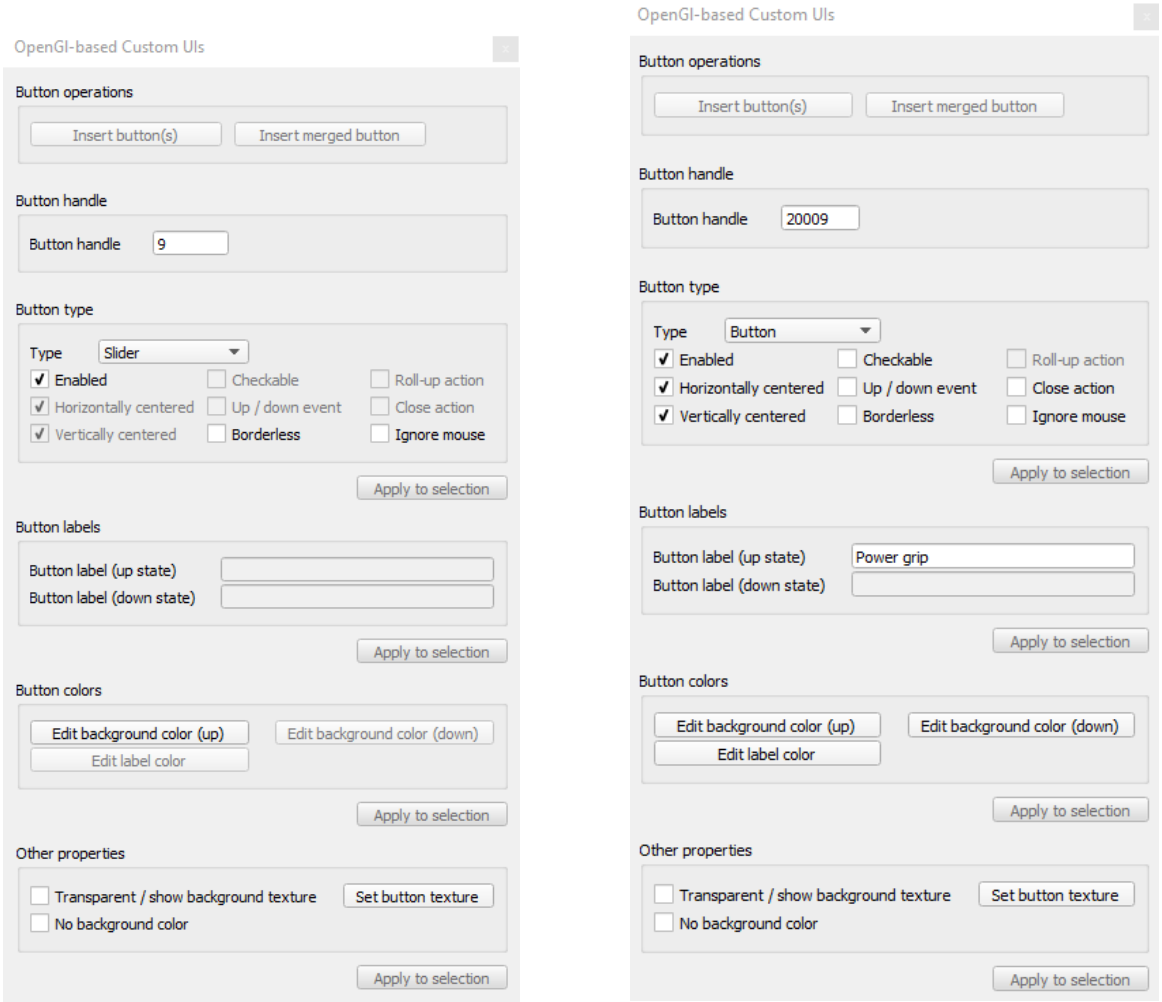

Ilustración 39, Configuración botón y slider

Con todo este proceso ya se ha diseñado correctamente la interfaz gráfica de la herramienta, quedando así preparada para el desarrollo del código encargado de la lectura de datos de la interfaz y su posterior procesado y uso en la simulación.

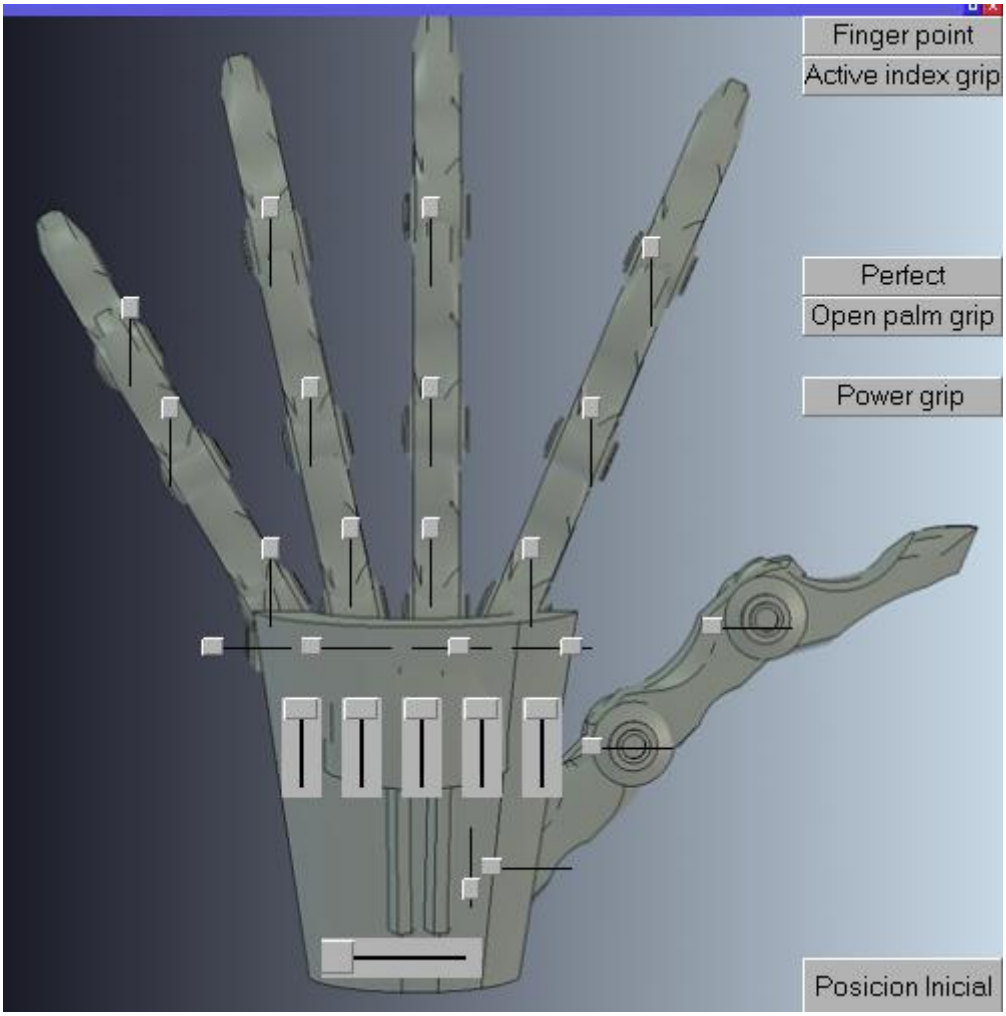

Ilustración 40, UI Inmoov-SR

Para el control de la mano desde la interfaz gráfica, se ha creado un Script (Anexo1: ActuateFingers).

Este script se ha generado en un elemento dummy auxiliar el cual se incluye en la jerarquía de la escena para poder guardar el código como parte del modelo.

En este script se ha programado en lenguaje LUA la simulación de movimiento en la herramienta Inmoov-SR mediante la variación de los sliders presentes en la interfaz gráfica.

Se va a explicar el código referente al citado script parte a parte, para conocer perfectamente el funcionamiento del mismo:

En primer lugar, ha sido creada una función llamada "setTarget", esta función hace referencia a los sliders grupales, es decir los cuales siendo modificados transmiten el movimiento a la totalidad de cada dedo.

En la siguiente imagen se puede observar el código referente a dicha función:

```
function setTarget(wrist,thumbgroup,indexgroup,middlegroup,ringgroup,pinkygroup)
    simSetUISlider(ui,8,thumbgroup)
    simSetUISlider(ui,9,indexgroup)
    simSetUISlider(ui,10,middlegroup)
    simSetUISlider(ui,11,ringgroup)
    simSetUISlider(ui,12,pinkygroup)
    actuate(wrist,thumbgroup,indexgroup,middlegroup,ringgroup,pinkygroup)
end
```

Ilustración 41, Código control herramienta

Como se puede observar, la función depende de la entrada de 6 parámetros, cada uno de ellos hace referencia al valor de cada momento de los sliders.

Mediante la función predeterminada "simSetUISliders" se introducen los valores pasados a la función al slider correspondiente, para ello, dicha función consta de tres entradas: ("nombre del UI", "Handler del slider", "valor a introducir")

Al final de esta función se llama a la función "actuate" introduciéndole los valores de los parámetros de entrada de la misma función.

Esta función es la encargada de configurar cada articulación de cada dedo al valor transferido en sus parámetros de entrada. En la siguiente imagen podemos observar como se ha realizado la programación para un dedo y la muñeca siendo el proceso y el código igual para cada uno de los dedos:

```
function actuate(wrist,pos1,pos2,pos3,pos4,pos5)
    local pos={wrist,pos1,pos2,pos3,pos4,pos5}
    local str=''
    if wrist~=nil then
        simSetUISlider(ui,7,wrist)
        str=str..string.format(' wrist %d',wrist)
    end
    if pos1~=nil then
        simSetUISlider(ui,10001, pos1 )
        simSetUISlider(ui,10002, pos1 )
        simSetUISlider(ui,10003, pos1 )
        str=str..string.format(' thumb %d',pos1)
    end
```

Ilustración 42,Código control herramienta

La parte principal de esta función como se puede ver es la desempeñada nuevamente por el comando "simSetUISlider" estando esta vez ejecutado por tres veces, es decir el número de articulaciones de cada dedo. Pero es importante observar que para que el movimiento sea uniforme e igual en todas ellas se le referencia al mismo valor, obtenido el cual de un array en el que se han almacenado los parámetros de entrada.

La siguiente función que se va a explicar se encarga del movimiento individual de cada articulación. Para este fin se distinguen dos partes principales en su código, la parte de obtención de datos*, encargada de leer la posición de los sliders y su posterior ajuste a un rango entre 0 y 90 grados almacenándolos en un array para cada dedo, y la parte de transmisión** de dichos datos al modelo de la herramienta para simular el movimiento. Esta función será llamada en el bucle principal del script ya que será necesario ejecutarla siempre, tanto si se han modificado los sliders individuales como los grupales será necesario enviar la posición al modelo simulado.

```
function actuateEachFingerJoints() --
    thumb[1]=(-(simGetUISlider(ui,10001) /1000 *90))*deg2rad --0 to 90 de
    thumb[2]=(-(simGetUISlider(ui,10002) /1000 *110))*deg2rad
    thumb[3]=(-(simGetUISlider(ui,10003) /1000 *90))*deg2rad

    index[1]=(-(simGetUISlider(ui,11001) /1000 *90))*deg2rad
    index[2]=(-(simGetUISlider(ui,11002) /1000 *110))*deg2rad
    index[3]=(-(simGetUISlider(ui,11003) /1000 *80))*deg2rad

    middle[1]=(-(simGetUISlider(ui,12001) /1000 *90))*deg2rad
    middle[2]=(-(simGetUISlider(ui,12002) /1000 *110))*deg2rad
    middle[3]=(-(simGetUISlider(ui,12003) /1000 *80))*deg2rad

    palmring= (-(simGetUISlider(ui,13000) /1000 *30))*deg2rad
    ring[1]= (-(simGetUISlider(ui,13001) /1000 *90))*deg2rad
    ring[2]= (-(simGetUISlider(ui,13002) /1000 *110))*deg2rad
    ring[3]= (-(simGetUISlider(ui,13003) /1000 *80))*deg2rad

    palmpinky=(-(simGetUISlider(ui,14000) /1000 *30))*deg2rad
    pinky[1]=(-(simGetUISlider(ui,14001) /1000 *90))*deg2rad
    pinky[2]=(-(simGetUISlider(ui,14002) /1000 *110))*deg2rad
    pinky[3]=(-(simGetUISlider(ui,14003) /1000 *80))*deg2rad

    wrist=(((simGetUISlider(ui,7)/1000)*180))*deg2rad
    simSetJointTargetPosition(hwrist,wrist)

    simSetJointTargetPosition(hpalmring,palmring)
    simSetJointTargetPosition(hpalmpinky,palmpinky)
    for i=1,3 do

        simSetJointTargetPosition(hthumb[i],thumb[i])
        simSetJointTargetPosition(hindex[i],index[i])
        simSetJointTargetPosition(hmiddle[i],middle[i])
        simSetJointTargetPosition(hring[i],ring[i])
        simSetJointTargetPosition(hpinky[i],pinky[i])
    end
end
```

*

**

Ilustración 43, Código control herramienta

Por otra parte, se ha realizado una parte de inicialización de las variables, en ella se han obtenido los handler de cada articulación que se puede modificar desde la interfaz, así como el handler de la UI.

Los handle de cada dedo se almacenan en un array para su posterior tratamiento explicado en la función anterior.

Es preciso también inicializar cada articulación a un valor fijado para que no surjan problemas al inicio de la simulación, en este caso la inicialización se realiza como la mano totalmente abierta.

Por último, se genera el bucle principal el cual se encarga de mantener actualizados los valores e ir transmitiéndoselos a las funciones encargadas de simular el movimiento de la mano.

En la siguiente imagen se recoge el código que forma dicho bucle:

```
171 while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) do
172     data=simGetStringSignal('actuate')
173     if data~=nil then
174         simClearStringSignal('actuate')
175         tg=simUnpackInts(data)
176         slideBackup=getAllMainSliders()
177         if tg~=nil then
178             for i=1,6 do
179                 if tg[i]==nil or tg[i]<0 or tg[i]>1000 then tg[i]=slideBackup[i] end
180             end
181             actuate(tg[1],tg[2],tg[3],tg[4],tg[5],tg[6])
182         end
183     end
184     actuateEachFingerJoints()
185 end
```

Ilustración 44, Código control herramienta

6.4.2 Control mediante Scripts empotrados.

El control mediante los scripts empotrados consta de varios elementos destacables, existen tres tipos de scripts según la funcionalidad que se les da para lograr la simulación, es necesaria la interacción con MATLAB mediante un fichero .m para la realización remota de movimientos (a través de la configuración de APIremota).

Todo script está asociado a un elemento dummy auxiliar los cuales están jerárquicamente ordenados dentro del modelo de la mano para garantizar la correcta ejecución de los mismos. Esto implica que exista un script principal "main" el cual recoge a todos los demás scripts en un orden de jerarquización menor.

Se va a comenzar explicando el código perteneciente al script "main" (Anexo2: Script main) encargado de la ejecución de los otros scripts de movimientos predeterminados dependiendo de la orden que se envíe desde los botones del interfaz UI.

Para la ejecución de un movimiento predeterminado mediante el interfaz grafico UI, es necesario primero identificar los manejadores de los botones los cuales nos ofrecen dicha posibilidad. Para ello se ha desarrollado unas líneas de código a través de las cuales se detecta un evento en cualquiera de los botones de acción del UI, guardando el valor del manejador de dicho botón en la variable "buttonID". También en este código se introduce la opción de que se varíen los sliders del UI en cuyo caso se da paso al script "actuateFingers" descrito en el anterior apartado.

El código que realiza estas funciones es el siguiente:

```
simHandleChildScripts(sim_call_type)

    buttonID=simGetUIEventButton(ui)

------Codigo para ajustar mediante los sliders del UI la posicion de los dedos------------

    if buttonID>=7 and buttonID<=12 then --6 main groups (5 fingers + wrist)
                                         -- actuate only one finger or wrist
        local targets={-1,-1,-1,-1,-1,-1}
        pos=simGetUISlider(ui,buttonID)
        targets[buttonID-6]=pos
        --to remove when using a proper function call
        for i=1,6 do
            targets[i]=simGetUISlider(ui,i+6)
        end
        --end to remove wwhen using proper function call
        fdata=simPackInts({targets[1],targets[2],targets[3],targets[4],targets[5],targets[6]})
        if fdata~=fdataold then
            --simSetStringSignal('actuate',fdata)
            simSetStringSignal('actuate',fdata)
            fdataold=fdata
        end
        predefMotion=false
    end
```

Ilustración 45, Script "main"

Una vez que se detecta una interacción en cualquiera de los botones del UI se da paso a las siguientes líneas de código, esto se hace gracias a un "if" que comprueba si el valor de la variable "buttonID" está comprendida entre los valores definidos como manejadores de los botones (manejadores entre el 20000 y el 20016).

Una vez que se entra en este apartado del código, el sistema comprueba a que botón corresponde la acción seleccionada y con correspondencia a los movimientos predeterminados que se han asignados a cada botón se manda la orden de ejecutar el script correspondiente a dicho movimiento.

```
146     if (buttonID >=20000 and buttonID<=20016) or cmd~=nil then
147         predefMotion=true
148 
149         --some signals cleanup and settings
150         simClearStringSignal('cmdSignal')
151 
152         -------- predefined position definition
153         if buttonID==20000 or cmd=='fingerPoint' then
154             sequence='fingerPoint'
155         end
156         if buttonID==20001 or cmd=='activeIndexGrip'then
157             sequence='activeIndexGrip'
158         end
159 
160         if buttonID==20006 or cmd=='mouseGrip'then
161             sequence='perfect'
162         end
163         if buttonID==20007 or cmd=='openPalmGrip' then
164             sequence='openPalmGrip'
165         end
166 
167         if buttonID==20009 or cmd=='powerGrip'then
168             sequence='powerGrip'
169         end
170 
171         if buttonID==20014 or cmd=='posicionInicial'then
172             sequence='posicionInicial'
173         end
174 
175 
176 
177     end
```

Ilustración 46, Script "main"

Mediante este método se accionan los movimientos predeterminados a través de los botones generados en el UI, pero es necesario también el desarrollo de unas líneas de código dentro del script “main para que sea posible la ejecución remota desde MATLAB de dichos movimientos predeterminados.

Hay que tener en cuenta que el siguiente código que se va a explicar recoge la parte necesaria en V-REP para la ejecución remota de las funciones desde MATLAB, pero no es el único requerimiento para ello, ya que más adelante se explicara el código desarrollado en MATLAB para poder realizar este fin.

Para la ejecución remota desde MATLAB se ha creado en el script “main” una función llamada “displayText_function” la cual tiene como parámetros de entrada un numero en formato int, un numero en formato float, un array y una cadena de caracteres, este último parámetro es el que vamos a utilizar para la recepción desde MATLAB del script a ejecutar.

En esta función, lo primero que se va a realizar es la comprobación de que la longitud de la cadena recibida, para ello mediante el operador "#" obtenemos la longitud de la cadena recibida, si la longitud es mayor que uno obtenemos la cadena almacenándola en la variable".

```
41 displayText_function=function(inInts,inFloats,inStrings,inBuffer)
42
43         if #inStrings>=1 then
44             x=inStrings[1]
45         end
46         if x=='fingerPoint' then
47             comando=x
48         end
49         if x=='activeIndexGrip' then
50             comando=x
51         end
52         if x=='openPalmGrip' then
53             comando=x
54         end
55         if x=='powerGrip' then
56             comando=x
57         end
58         if x=='posicionInicial' then
59             comando=x
60         end
61         if x=='perfect' then
62             comando=x
63         end
64         if x=='PosicionDeterminada' then
65
66             comando=x
67         end
68
```

Ilustración 47, Script "main"

Una vez comprobado el nombre del script recibido se almacena en la variable "comando" siendo entonces cuando según el nombre guardado se manda ejecutar el script correspondiente.

En la siguiente imagen se observa cómo se realizaría el llamamiento al script para uno de los casos, siendo el mismo proceso para el resto de scripts.

```
 99         if comando=='PosicionDeterminada' then
100             predefMotion=true
101             sequence='PosicionDeterminada'
102         end
103
104         return {},{},{'Finalizado'},''
```

Ilustración 48, Script "main"

Ahora se va a explicar el modelo de script creado para realizar cada uno de los movimientos predeterminados.

En primer lugar, se han de fijar el valor de las variables las cuales nos establecen la posición de cada articulación, estos valores oscilaran entre 0 y 973 para un movimiento optimo sin llegar a colisionar con la palma de la mano, siendo 0 la posición correspondiente al dedo extendido y 973 al dedo cerrado.

En la siguiente imagen se observa el código necesario para dichas inicializaciones:

```
    phase={    tempo=   {0.8 }, -- timeout on actions if -1 then no timeout but trigger action
            pnext=     {2   }, -- pointer to the next index
            twrist=  {0   },
            tthumb=  {posicion_pulgar},
            tindex=  {posicion_indice},
            tmiddle= {posicion_medio },
            tring=   {posicion_anular },
            tpinky=  {posicion_menique}
            }
    --sequence intitialization
    index=1
    indexNum=#phase.tempo
    starttime=simGetSystemTime() --start action timer
```

Ilustración 49, Script posición determinada

Las variables posición ”dedo” marca las posiciones mencionadas, en los scripts configurados para movimientos predeterminados estas variables tienen en el mismo script fijado su valor como se puede ver en los anexos que se incluyen dichos códigos.

En el caso mostrado dichas variables serán enviadas desde Matlab gracias a funciones de la api remota, dicha función y la forma de transmisión serán explicadas más adelante.

El resto de código de los scripts generadores de movimiento son comunes para todos los scripts. Esta zona se encarga de comprobar la posición actual de los dedos y modificarla dentro del bucle hasta que se adapta a los valores introducidos en la constante “phase”.

Como se ha mencionado anteriormente en el script “posiciónDeterminada“ recibe de forma remota los valores para ello son necesarias al comienzo del script las siguientes líneas:

```
--------------------------------------------------------------------------
local signal_array1=simGetStringSignal('transmision1')

if signal_array1 then
    posicion_dedos=simUnpackInts(signal_array1)
end
if posicion_dedos~=nil then
            posicion_pulgar=posicion_dedos[1]
            posicion_indice=posicion_dedos[2]
            posicion_medio=posicion_dedos[3]
            posicion_anular=posicion_dedos[4]
            posicion_menique=posicion_dedos[5]
end
```

Ilustración 50, Script posición determinada

Gracias a la función “simGetStringSignal” se recibe la transmisión enviada desde la función de MATLAB “llamada_Funcion” (Anexo: Función “llamada_Funcion”) y mediante el uso de “simUnpackInts” se descomprime el paquete de datos enviados almacenando en un array el array enviado.

Dichas funciones las podemos encontrar en el manual de usuario de V-REP en el apartado funciones de la API remota [10] y vienen descritas de la siguiente forma:

- simGetStringSignal y simSetStringSignal

simxGetStringSignal (regular API equivalent: simGetStringSignal)

Description	Gets the value of a string signal. Signals are cleared at simulation start. See also simxSetStringSignal, simxReadStringStream, simxClearStringSignal, simxGetIntegerSignal and simxGetFloatSignal.
C synopsis	simxInt simxGetStringSignal(simxInt clientID,const simxChar* signalName,simxUChar** signalValue,simxInt* signalLength,simxInt operationMode)
C parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **signalValue**: pointer to a pointer receiving the value of the signal. The signal value will remain valid until next remote API call **signalLength**: pointer to a location receiving the value of the signal length, since it may contain any data (also embedded zeros). **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
C return value	a remote API function return code
Other languages	Python, Java, Matlab, Octave, Urbi, Lua

simxSetStringSignal (regular API equivalent: simSetStringSignal)

Description	Sets the value of a string signal. If that signal is not yet present, it is added. See also simxWriteStringStream, simxGetStringSignal, simxClearStringSignal, simxSetIntegerSignal and simxSetFloatSignal.
C synopsis	simxInt simxSetStringSignal(simxInt clientID,const simxChar* signalName,const simxUChar* signalValue,simxInt signalLength,simxInt operationMode)
C parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **signalValue**: value of the signal (which may contain any value, including embedded zeros) **signalLength**: size of the signalValue string. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
C return value	a remote API function return code
Other languages	Python, Java, Matlab, Octave, Urbi, Lua

Ilustración 51, Función API remota GET/SET string

- simPackInts y simunPackInts

simxUnpackInts

Description	Unpacks a string into an array of integers. This is a remote API helper function. See also simxPackInts and simxUnpackFloats.
Python synopsis	array intValues=simxUnpackInts(string packedData)
Python parameters	**packedData**: a string that contains the packed values. Each values takes exactly 4 bytes in the string.
Python return values	**intValues**: an array of numbers that were unpacked as integers
Other languages	Java, Matlab, Octave, Urbi, Lua

simxPackInts

Description	Packs an array of integers into a string. This is a remote API helper function. See also simxUnpackInts and simxPackFloats.
Python synopsis	string packedData=simxPackInts(array intValues)
Python parameters	**intValues**: an array of numbers we wish to pack as integers
Python return values	**packedData**: a string that contains the packed values. Each values takes exactly 4 bytes in the string.
Other languages	Java, Matlab, Octave, Urbi, Lua

Ilustración 52, Función API remota, pack/unpack

De esta forma queda explicado el método utilizado para el desarrollo de los scripts en V-REP.

Para finalizar la parte experimental, queda explicar la función creada en MATLAB para la comunicación entre las dos plataformas y la correcta ejecución de los movimientos predeterminados de forma remota.

Dicha función consta de dos parámetros de entrada, el primero de ellos se corresponde con una cadena de caracteres a través de la cual introduciremos el nombre del script el cual queramos ejecutar; en lo correspondiente al segundo parámetro es un buffer de enteros, los cuales serán la configuración de cada dedo que le queremos transmitir a la mano, esto hace que este parámetro solo sea necesario introducirlo en el caso que se ejecute el script "posicionDeterminada" ya que el resto de scripts ya incluyen estos valores por defecto.

En primer lugar, creamos la conexión con el servidor remoto API :

```
ini_robotics; %Peter Corke library call
disp('Program started');
vrep=remApi('remoteApi'); % using the prototype file
vrep.simxFinish(-1); % just in case, close all opened connections
clientID=vrep.simxStart('127.0.0.1',19999,true,true,5000,5);
```

A continuación, trasmitimos el buffer introducido mediante las funciones que nos ofrece la Api remota:

```
posiciones=buffer;
pos1=vrep.simxPackInts(posiciones)
vrep.simxSetStringSignal(clientID,'transmision1',pos1,vrep.simx_
opmode_streaming);
```

Por último, falta enviar la señal para ejecutar el script "main" de V-REP, y dentro de él ejecutar la función descrita con anterioridad a través de la cual y el parámetro de entrada con el nombre del movimiento a realizar este será ejecutado remotamente:

```
[res retInts retFloats retStrings
retBuffer]=vrep.simxCallScriptFunction(clientID,'main',vrep.sim_script
type_childscript,'displayText_function',[],[],accion,buffer,vrep.simx_
opmode_blocking);
```

Esta función queda definida dentro de las funciones de la API remota [10] de la siguiente forma:

simxCallScriptFunction (regular API equivalent: simCallScriptFunctionEx)

Description	Remotely calls a V-REP script function. When calling simulation scripts, then simulation must be running (and threaded scripts must still be running, i.e. not ended yet). Refer to this section for additional details.
C synopsis	simxInt simxCallScriptFunction(simxInt clientID,const simxChar* scriptDescription,simxInt scriptHandleOrType,const simxChar* functionName,simxInt inIntCnt,const simxInt* inInt,simxInt inFloatCnt,const simxFloat* inFloat,simxInt inStringCnt,const simxChar* inString,simxInt inBufferSize,const simxUChar* inBuffer,simxInt* outIntCnt,simxInt** outInt,simxInt* outFloatCnt,simxFloat** outFloat,simxInt* outStringCnt,simxChar** outString,simxInt* outBufferSize,simxUChar** outBuffer,simxInt operationMode)
C parameters	**clientID**: the client ID. refer to simxStart. **scriptDescription**: the name of the scene object where the script is attached to, or an empty string if the script has no associated scene object. **scriptHandleOrType**: the handle of the script, otherwise the type of the script: *sim_scripttype_mainscript* (0): the main script will be called. *sim_scripttype_childscript* (1): a child script will be called. *sim_scripttype_jointctrlcallback* (4): a joint control callback script will be called. *sim_scripttype_contactcallback* (5): the contact callback script will be called. *sim_scripttype_customizationscript* (6): a customization script will be called. *sim_scripttype_generalcallback* (7): the general callback script will be called. **functionName**: the name of the Lua function to call in the specified script. **inIntCnt** (input): the number of input integer values. **inInt** (input): the input integer values that are handed over to the script function. Can be NULL if *inIntCnt* is zero. **inFloatCnt** (input): the number of input floating-point values. **inFloat** (input): the input floating-point values that are handed over to the script function. Can be NULL if *inFloatCnt* is zero. **inStringCnt** (input): the number of input strings. **inString** (input): the input strings that are handed over to the script function. Each string should be terminated with one zero char, e.g. "Hello\0World\0". Can be NULL if *inStringCnt* is zero. **inBufferSize** (input): the size of the input buffer. **inBuffer** (input): the input buffer (bytes) that is handed over to the script function. Can be NULL if *inBufferSize* is zero. **outIntCnt** (output): the number of returned integer values. Can be NULL. **outInt** (output): the returned integer values. The pointer remains valid until the next remote API call. Can be NULL. **outFloatCnt** (output): the number of returned floating-point values. Can be NULL. **outFloat** (output): the returned floating-point values. The pointer remains valid until the next remote API call. Can be NULL. **outStringCnt** (output): the number of returned strings. Can be NULL. **outString** (output): the returned strings. Each string is terminated with the zero char. The pointer remains valid until the next remote API call. Can be NULL. **outBufferSize** (output): the size of the returned buffer. Can be NULL. **outBuffer** (output): the returned buffer (bytes). The pointer remains valid until the next remote API call. Can be NULL. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
C return value	a remote API function return code
Other languages	Python, Java, Matlab, Octave, Urbi, Lua

Ilustración 53, Función API remota ejecutar función

Por último, se termina la función deteniendo la conexión para que el programa quede preparado para la siguiente función.

7. Resultados.

Todas las escenas, modelos y archivos de MATLAB se encuentran en los archivos digitales adjuntos

Se ha llegado a realizar un modelado muy preciso tanto del brazo robótico como de la herramienta Inmoov-SR, como se puede observar en las siguientes imágenes:

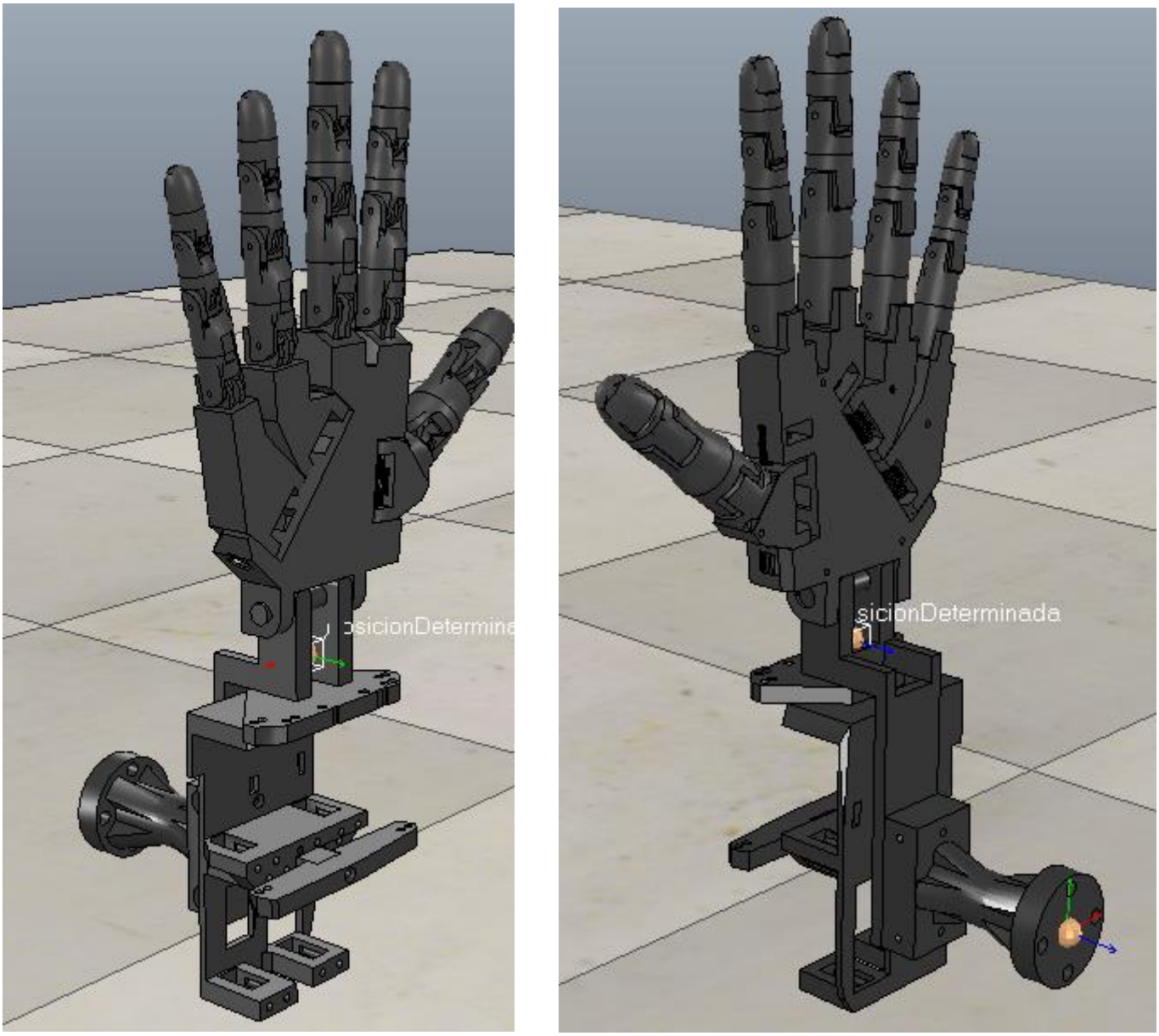

Ilustración 54, INMOOV-SR resultado

Se puede apreciar el gran detalle que nos proporciona importar la mano desde un archivo generado con un software de diseño 3D.

El archivo obtenido del brazo robótico IRB120 también ofrece una alta calidad como se aprecia a continuación:

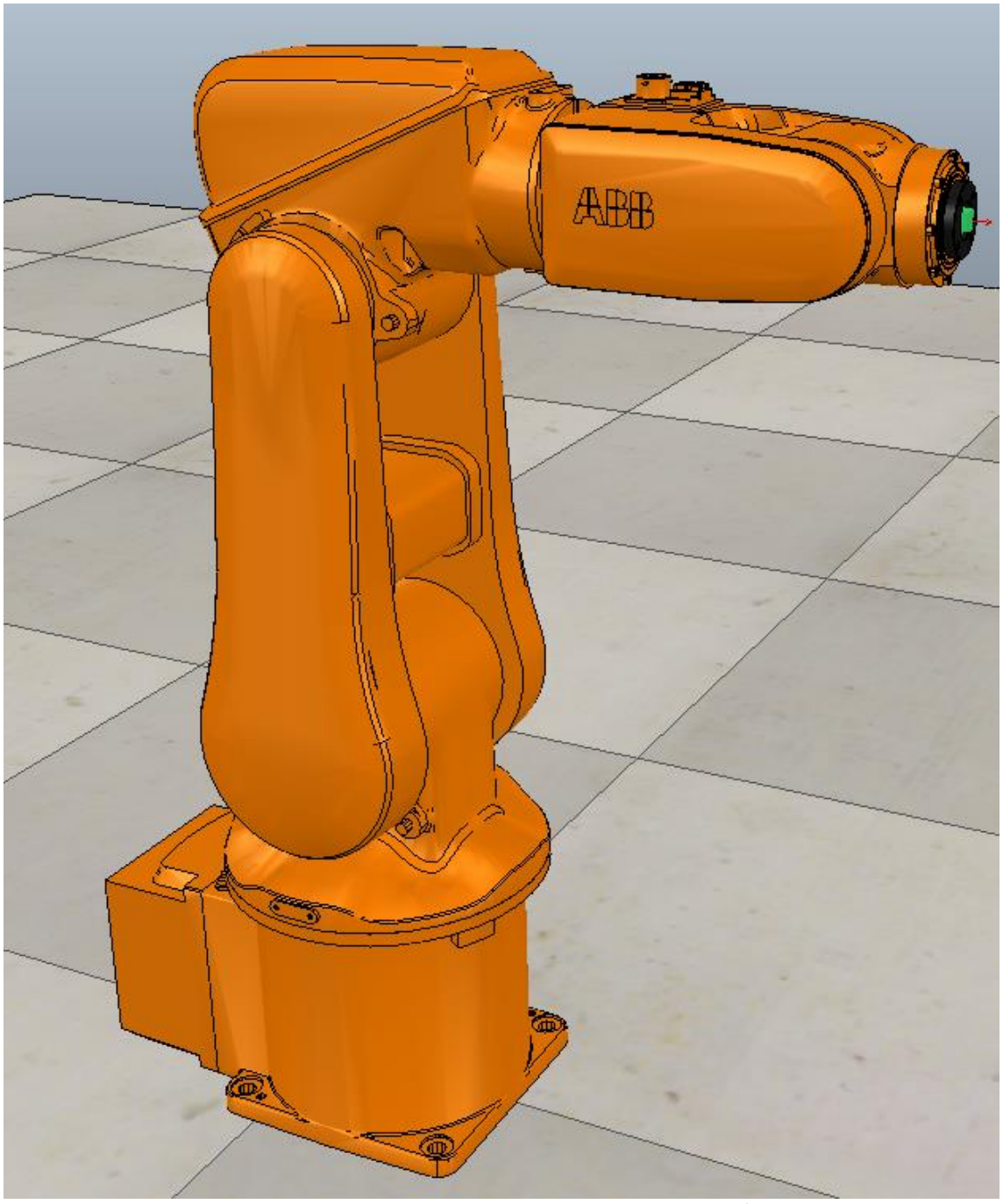

Ilustración 55, IRB120 resultado

Tambien la configuracion del objeto a simulado perfectamente la inercia y configuracion del robot.

El proceso de control ha sido muy satisfactorio, ya que se ha conseguido realizar un control optimo del robot desde el software MATLAB al realizar la conexión entre ambas plataformas con la ayuda de la toolbox de robótica de Peter Corke, así como de las funciones de la API remota.

Se ha conseguido también la implementación de los script de control de la mano a través de los cuales se han configurado unos movimientos predeterminados los cuales pueden ser accionados desde los botones incluidos en la interfaz gráfica, así como siendo accionado desde MATLAB mediante él envió de una cadena de caracteres.

La interfaz gráfica ha sido de otra forma configurada para poder determinar la posición de cada dedo gracias a los sliders colocados en cada articulación.

Un script ha sido configurado de forma específica para recibir desde MATLAB un buffer de 5 valores enteros siendo cada uno de ellos la configuración a tomar por cada dedo.

Por último, se ha acoplado la herramienta al brazo robótico y montado en una estación de trabajo similar a la del laboratorio de robótica de la Universidad de Alcalá, simulando que todo lo descrito funcionara y se pudiera simular cualquier situación.

La escena final obtenida es la representada en la siguiente imagen:

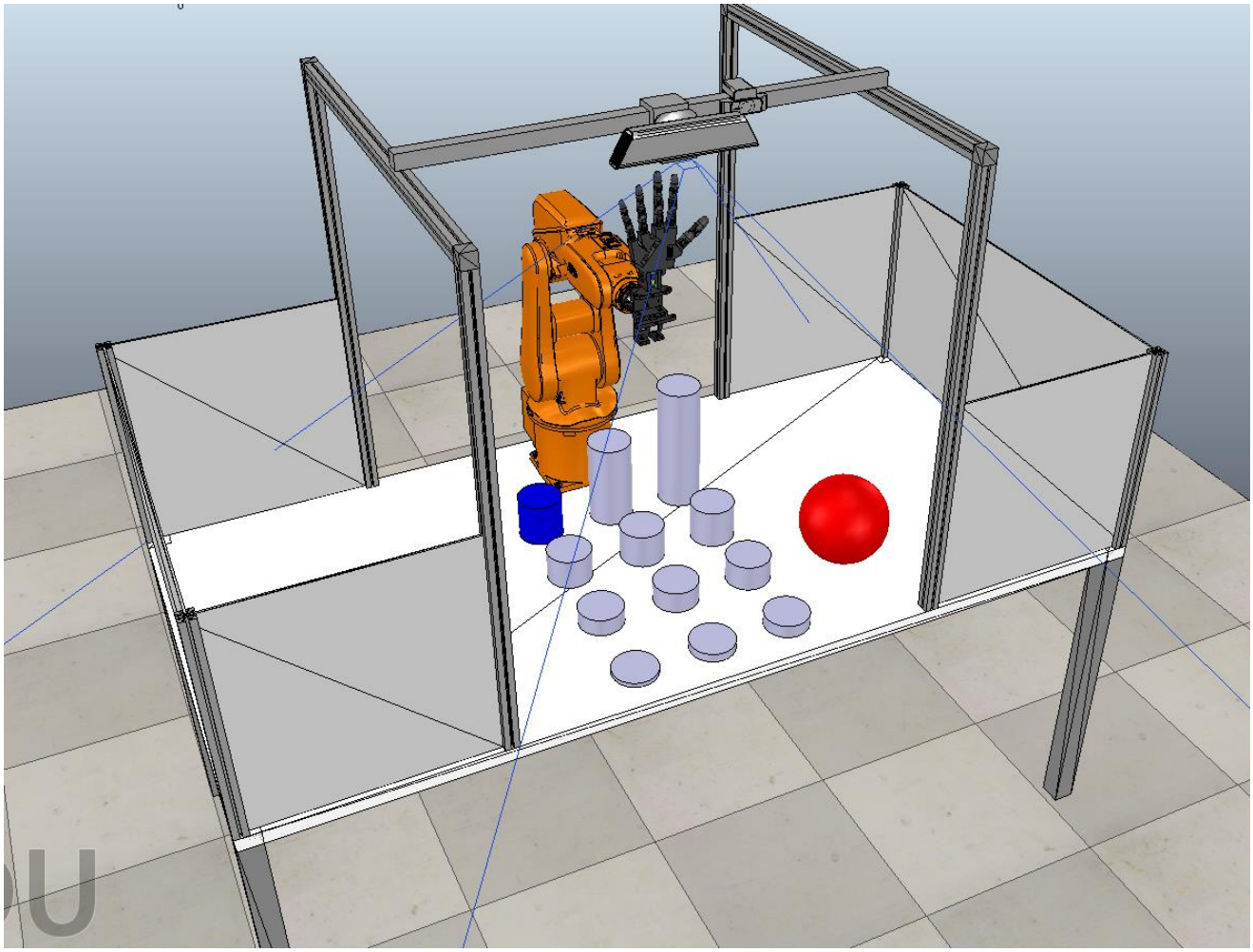

Ilustración 56, Estación final de trabajo

8. Conclusiones y trabajo futuro.

En este proyecto se ha observado todo el potencial que desarrolla el entorno de simulación de V-REP incluso en niveles de aprendizaje como es este caso, ya que se comenzó a realizar el trabajo sin referencias sobre el programa y se ha terminado manejando con soltura para simulación de una dificultad media.

Esto se debe en gran medida a la información existente en internet sobre el mismo, ya que la empresa desarrolladora, Coppelia, ha puesto a nuestra disposición gran cantidad de ayudas para un aprendizaje más rápido.

Al igual que Coppelia hay que destacar la colaboración en múltiples páginas web y entidades mencionadas anteriormente y referenciadas a la bibliografía, ya que gracias a ellas se han podido obtener los modelos necesarios, información sobre el control y una gran cantidad de ejemplos.

A todo aquel que vaya a comenzar con el uso de este entorno de simulación, se le recomienda el estudio de las múltiples escenas de prueba instaladas por defecto con el programa, ya que ayudan de sobremanera a entender el funcionamiento del mismo, de la misma forma que una participación en el foro de V-REP [3] ya que los administradores y desarrolladores están siempre disponibles para la resolución de problemas y generalmente leyendo comentarios de otros usuarios facilitan un correcto uso.

Hemos quedado muy gratamente sorprendidos con la facilidad que se realiza la conexión remota entre MATLAB y V-REP, así como la gran cantidad de funciones que se ofrecen tanto en la API remota como en la Toolbox de robótica de Peter Corke.

Si es cierto que el código a realizar en MATLAB se ha escrito con mayor facilidad, debido a que es un lenguaje más estudiado a lo largo de la carrera. En el caso del script de V-REP se ha hecho algo más tedioso por la necesidad de adaptarnos al lenguaje LUA, nunca visto con anterioridad, y a las funciones que se usan.

Finalmente se consiguió un modelado con gran precisión de los dos objetos principales, el brazo robótico IRB120 y la herramienta Inmoov-SR. De la misma forma conseguimos realizar un control de forma remota del IRB120 desde MATLAB, y un control mediante una interfaz gráfica y scripts empotrados en V-REP de la herramienta, añadiendo además la posibilidad de accionar dichos scripts desde MATLAB o incluso mandar de forma individual la posición a cada dedo.

8.1 Trabajo futuro.

Se ha fijado como trabajo futuro la mejora del modelo y control de la herramienta mediante la inserción de sensores de fuerza en las articulaciones a través de los cuales se podría simular de forma más realista el cierre de la mano ya que se determinaría por la fuerza ejercida por cada dedo.

Esta mejora permitiría coger cualquier tipo de objetos en la simulación, manteniéndolos unido a la mano por la propia fuerza ejercida por ella.

Pliego de condiciones

En este apartado se muestran los requisitos de hardware y software que han sido necesarios para la realización de este proyecto.

1. Requisitos de hardware.

- Ordenador personal (portátil) ASUS GL552VW
  - Procesador Intel Core i7-6700HQ
  - 20Gb de memoria RAM
  - Sistema operativo de 64 bits
  - CPU de 2.6GHz.
- Brazo robótico
  - ABB IRB120
- Herramienta
  - Inmoov-SR

2. Requisitos de Software.

- Requisitos generales:
  - Sistema operativo Windows 10

- Requisitos específicos.
  - MATLAB incluyendo la Toolbox de robótica de Peter Corke.
  - V-REP

- Editores de texto:
  - Word 2016
  - Adobe Acrobat Reader DC.
  - PowerPoint

Presupuesto

Se han dividido el coste total de la siguiente forma:

- Costes de Software

Recursos software					
Concepto	**Observaciones**	**Coste**	**Amortizacion (años)**	**Tiempo de uso (meses)**	**TOTAL**
MATLAB r2016B	Licencia universitaria	0 €	N/A	8	0 €
V-REP	Licencia educativa	0 €	N/A	8	0 €
WORD 2016	Licencia universitaria	0 €	N/A	5	0 €
Adobe Reader DC		0 €	N/A	8	0 €
TOTAL					0 €

Ilustración 57, Costes Software

- Coste de hardware.

Recursos hardware					
Concepto	**Observaciones**	**Coste**	**Amortizacion (años)**	**Tiempo de uso (meses)**	**TOTAL**
ABB IRB120		20.000 €	4	8	3.333 €
INMOOV-SR	Material y montaje	200 €	4	8	33 €
ASUS GL552VW		1.400 €	4	6	175 €
AlienWare M15X		1.550 €	4	2	65 €
TOTAL					3.606 €

Ilustración 58, Costes hardware

- Mano de obra.

Mano de obra			
Concepto	**Coste por hora**	**Horas**	**Total**
Montaje Inmoov-SR	10€/h	20	200 €
Ejecución	15€/h	200	3.000 €
Redacción	10€/h	100	1.000 €
TOTAL			4.200 €

Ilustración 59, Costes mano de obra

A partir de los datos anteriormente reflejados, se procede a obtener el coste total del proyecto. El conste de desarrollo por una contrata sigue la siguiente ecuación:

$$PEC = CM + b_{\%} * CM$$

Donde el PEC es el presupuesto total, el CM es el coste material de ejecución tomado por la suma de las cuantías totales expuestas con anterioridad y el $b_{\%}$ es el porcentaje del material como beneficio industrial (entorno al 30%).

Siguiendo esta fórmula obtenemos:

$$PEC = (3606 + 4200) + 0.3 * 7806 = 10147.8€$$

El coste final del proyecto es de DIEZ MIL CIENTO CUARENTA Y SIETE EUROS CON OCHENTA CENTIMOS.

Bibliografía

[1] Wikipedia, «Wikipedia,» 28 Junio 2017. [En línea]. Available: https://es.wikipedia.org/wiki/Rob%C3%B3tica.

[2] Coppelia, «Coppelia,» [En línea]. Available: http://www.coppeliarobotics.com/helpFiles/.

[3] C. Forum, «Coppelia Forum,» [En línea]. Available: http://www.forum.coppeliarobotics.com/.

[4] Coppelia, «Funciones Api remota,» [En línea]. Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctions.htm#simxGetStringSignal.

[5] P. Corke, «Robotics ToolBox,» [En línea]. Available: http://petercorke.com/wordpress/toolboxes/robotics-toolbox.

[6] Github, «GitHub,» URDF files, [En línea]. Available: https://github.com/evenator/swri-ros-pkg/tree/master/abb/abb_common/urdf.

[7] G. Langevin, «inmoov.fr,» [En línea]. Available: http://inmoov.fr/.

[8] E. Rohmer, «Universidad de Galileo,» Turing Laboratory, [En línea]. Available: http://turing-lab.github.io/.

[9] L. Nieto Fabian y E. Guillermo, 2016.

[10] A. G. Corbacho, Desarrollo de una interfaz para el control del robot IRB120 desde MATLAB, 2014.

Anexo 1: Script "actuateFingers"

```
-- actuate the grouped joints (e.g. 6 main sliders)
-- TODO modify signal for function call when integrated in future V-REP
simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()
function setTarget(wrist,thumbgroup,indexgroup,middlegroup,ringgroup,pinkygroup)
    simSetUISlider(ui,8,thumbgroup)
    simSetUISlider(ui,9,indexgroup)
    simSetUISlider(ui,10,middlegroup)
    simSetUISlider(ui,11,ringgroup)
    simSetUISlider(ui,12,pinkygroup)
    actuate(wrist,thumbgroup,indexgroup,middlegroup,ringgroup,pinkygroup)
end


function actuate(wrist,pos1,pos2,pos3,pos4,pos5) -- set each individual sliders according to
groups
    local pos={wrist,pos1,pos2,pos3,pos4,pos5}
    local str=""
    if wrist~=nil then simSetUISlider(ui,7,wrist)
        str=str..string.format(" wrist %d",wrist)
    end
    if pos1~=nil then simSetUISlider(ui,10001,
        pos1 ) simSetUISlider(ui,10002, pos1 )
        simSetUISlider(ui,10003, pos1 )
        str=str..string.format(" thumb %d",pos1)
    end
    if pos2~=nil then simSetUISlider(ui,11001,
        pos2 ) simSetUISlider(ui,11002, pos2 )
        simSetUISlider(ui,11003, pos2 )
        str=str..string.format(" index %d",pos2)
    end
    if pos3~=nil then simSetUISlider(ui,12001,
        pos3 ) simSetUISlider(ui,12002, pos3 )
        simSetUISlider(ui,12003, pos3 )
        str=str..string.format(" middle %d",pos3)
    end
    if pos4~=nil then simSetUISlider(ui,13001,
        pos4 ) simSetUISlider(ui,13002, pos4 )
        simSetUISlider(ui,13003, pos4 )
        str=str..string.format(" ring  %d",pos4)
    end
    if pos5~=nil then simSetUISlider(ui,14001,
        pos5 ) simSetUISlider(ui,14002, pos5 )
        simSetUISlider(ui,14003, pos5 )
        str=str..string.format(" pinky %d",pos5)
    end

    for i=1,6 do
        if pos[i]~=nil then simSetUISlider(ui,i+6,pos[i]) end
    end

    --simAddStatusbarMessage('in actuate()'..str)
    predefMotion=false
end


function actuateEachFingerJoints() -- actuate each fingers jonts and the wrist depending on
their individual slider

    thumb[1]=(-(simGetUISlider(ui,10001) /1000 *90))*deg2rad --0 to 90 deg
    thumb[2]=(-(simGetUISlider(ui,10002) /1000 *110))*deg2rad
    thumb[3]=(-(simGetUISlider(ui,10003) /1000 *90))*deg2rad

    index[1]=(-(simGetUISlider(ui,11001) /1000 *90))*deg2rad
```

```
    index[2]=(-(simGetUISlider(ui,11002) /1000 *110))*deg2rad
    index[3]=(-(simGetUISlider(ui,11003) /1000 *80))*deg2rad

    middle[1]=(-(simGetUISlider(ui,12001) /1000 *90))*deg2rad
    middle[2]=(-(simGetUISlider(ui,12002) /1000 *110))*deg2rad
    middle[3]=(-(simGetUISlider(ui,12003) /1000 *80))*deg2rad

    palmring= (-(simGetUISlider(ui,13000) /1000 *30))*deg2rad
    ring[1]= (-(simGetUISlider(ui,13001) /1000 *90))*deg2rad
    ring[2]= (-(simGetUISlider(ui,13002) /1000 *110))*deg2rad
    ring[3]= (-(simGetUISlider(ui,13003) /1000 *80))*deg2rad

    palmpinky=(-(simGetUISlider(ui,14000) /1000 *30))*deg2rad
    pinky[1]=(-(simGetUISlider(ui,14001) /1000 *90))*deg2rad
    pinky[2]=(-(simGetUISlider(ui,14002) /1000 *110))*deg2rad
    pinky[3]=(-(simGetUISlider(ui,14003) /1000 *80))*deg2rad

    wrist=(((simGetUISlider(ui,7)/1000)*180))*deg2rad
    simSetJointTargetPosition(hwrist,wrist)

    simSetJointTargetPosition(hpalmring,palmring)
    simSetJointTargetPosition(hpalmpinky,palmpinky)
    for i=1,3 do
    --simAddStatusbarMessage(string.format("thumb[%d]: %f",i,thumb[i]*rad2deg))
        simSetJointTargetPosition(hthumb[i],thumb[i])
        simSetJointTargetPosition(hindex[i],index[i])
        simSetJointTargetPosition(hmiddle[i],middle[i])
        simSetJointTargetPosition(hring[i],ring[i])
        simSetJointTargetPosition(hpinky[i],pinky[i])
    end
end


function getAllMainSliders()
    mainSlidersBackup={}
    mainSlidersBackup[1]=simGetUISlider(ui,7)
    mainSlidersBackup[2]=simGetUISlider(ui,8)
    mainSlidersBackup[3]=simGetUISlider(ui,9)
    mainSlidersBackup[4]=simGetUISlider(ui,10)
    mainSlidersBackup[5]=simGetUISlider(ui,11)
    mainSlidersBackup[6]=simGetUISlider(ui,12)
    return mainSlidersBackup
end

--#########################  Initialization
deg2rad=math.pi/180
rad2deg=180/math.pi
ui=simGetUIHandle("UI")
hwrist=simGetObjectHandle("wrist")
wrist= 0 * deg2rad

--hoppothumb=simGetObjectHandle("thumb0")
--oppothumb= 0 * deg2rad

hpalmring=simGetObjectHandle("palm_ring")
hpalmpinky=simGetObjectHandle("palm_pinky")

hthumb={}
hthumb[1]=simGetObjectHandle("thumb1")
hthumb[2]=simGetObjectHandle("thumb2")
hthumb[3]=simGetObjectHandle("thumb3")

thumb={}
thumb[1]=  45*deg2rad
thumb[2]=  45*deg2rad
thumb[3]= 45*deg2rad

hindex={}
hindex[1]=simGetObjectHandle("index1")
hindex[2]=simGetObjectHandle("index2")
```

```
hindex[3]=simGetObjectHandle("index3")

index={}
index[1]=45*deg2rad
index[2]=45*deg2rad
index[3]=45*deg2rad

hmiddle={}
hmiddle[1]=simGetObjectHandle("middle1")
hmiddle[2]=simGetObjectHandle("middle2")
hmiddle[3]=simGetObjectHandle("middle3")

middle={}
middle[1]=45*deg2rad
middle[2]=45*deg2rad
middle[3]=45*deg2rad

hring={}
hring[1]=simGetObjectHandle("ring1")
hring[2]=simGetObjectHandle("ring2")
hring[3]=simGetObjectHandle("ring3")
ring={}
ring[1]=45*deg2rad
ring[2]=45*deg2rad
ring[3]=45*deg2rad
hpinky={}
hpinky[1]=simGetObjectHandle("pinky1")
hpinky[2]=simGetObjectHandle("pinky2")
hpinky[3]=simGetObjectHandle("pinky3")
pinky={}
pinky[1]=45*deg2rad
pinky[2]=45*deg2rad
pinky[3]=45*deg2rad

--################# Main loop
while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) do
    data=simGetStringSignal('actuate')
    if data~=nil then
--print('in condition actuate',simGetSystemTime())
        simClearStringSignal('actuate')
        tg=simUnpackInts(data)
        slideBackup=getAllMainSliders()
        if tg~=nil then
            for i=1,6 do
                if tg[i]==nil or tg[i]<0 or tg[i]>1000 then tg[i]=slideBackup[i] end
            end
            actuate(tg[1],tg[2],tg[3],tg[4],tg[5],tg[6])
        end
    end

    actuateEachFingerJoints()
end

actuate(0,0,0,0,0,0)
```

Anexo 2: Script "main"

```
-----------------------------------------------------------------------------
-- Following few lines automatically added by V-REP to guarantee compatibility
-- with V-REP 3.1.3 and later:
if (sim_call_type==sim_childscriptcall_initialization) then

    simSetScriptAttribute(sim_handle_self,sim_childscriptattribute_automaticcascadingcalls,fal
    se)
end

if (sim_call_type==sim_childscriptcall_cleanup) then

end
if (sim_call_type==sim_childscriptcall_sensing) then
    simHandleChildScripts(sim_call_type)
end
if (sim_call_type==sim_childscriptcall_actuation) then
    if not firstTimeHere93846738 then
        firstTimeHere93846738=0
    end

    simSetScriptAttribute(sim_handle_self,sim_scriptattribute_executioncount,firstTimeHere9384
    6738)
    firstTimeHere93846738=firstTimeHere93846738+1

-----------------------------------------------------------------------------

if (simGetScriptExecutionCount()==0) then

        deg2rad=math.pi/180
        rad2deg=180/math.pi
        ui=simGetUIHandle("UI")
        --ui=simGetUIHandle('UItmp')
        a1bt=3 a2bt=4
        trigbt=6
        nextbt=7
        quitbt=8
        notpredefbt=5
        predef=false

        mainHandle=simGetObjectAssociatedWithScript(sim_handle_self)
end

displayText_function=function(inInts,inFloats,inStrings,inBuffer)

        if #inStrings>=1 then
            x=inStrings[1]
        end
        if x=='fingerPoint' then
            comando=x
        end
        if x=='activeIndexGrip' then
            comando=x
        end
        if x=='mouseGrip' then
            comando=x
        end
        if x=='openPalmGrip' then
            comando=x
        end
        if x=='powerGrip' then
            comando=x
        end
        if x=='posicionInicial' then
            comando=x
        end
        if x=='perfect' then
            comando=x
```

```
        end
        if x=='PosicionDeterminada' then

            comando=x
        end



        if comando~=nil then
            simAddStatusbarMessage(comando..'')
        end


        ------------------------------------------------------------------------------------
        ----------------------------------
        if comando=='fingerPoint' then
            predefMotion=true
            sequence='fingerPoint'
        end
        if comando=='activeIndexGrip' then
            predefMotion=true
            sequence='activeIndexGrip'
        end
        if comando=='mouseGrip' then
            predefMotion=true
            sequence='mouseGrip'
        end
        if comando=='openPalmGrip' then
            predefMotion=true
            sequence='openPalmGrip'
        end
        if comando=='powerGrip' then
            predefMotion=true
            sequence='powerGrip'
        end
        if comando=='posicionInicial' then
            predefMotion=true
            sequence='posicionInicial'
        end
        if comando=='perfect' then
            predefMotion=true
            sequence='perfect'
        end
        if comando=='PosicionDeterminada' then
            predefMotion=true
            sequence='PosicionDeterminada'
        end

        return {},{},{'Finalizado'},''
    end
if (sim_call_type==sim_childscriptcall_initialization) then
    simExtRemoteApiStart(19999)
end
 simHandleChildScripts(sim_call_type)

    buttonID=simGetUIEventButton(ui)

------Codigo para ajustar mediante los sliders del UI la posicion de los dedos-----------

    if buttonID>=7 and buttonID<=12 then --6 main groups (5 fingers + wrist)
                                         -- actuate only one finger or wrist
        local targets={-1,-1,-1,-1,-1,-1}
        pos=simGetUISlider(ui,buttonID)
        targets[buttonID-6]=pos
        --to remove when using a proper function call
        for i=1,6 do
            targets[i]=simGetUISlider(ui,i+6)
        end
        --end to remove wwhen using proper function call
```

```
fdata=simPackInts({targets[1],targets[2],targets[3],targets[4],targets[5],targets[6]})

        if fdata~=fdataold then
            --simSetStringSignal('actuate',fdata)
            simSetStringSignal('actuate',fdata)
            fdataold=fdata
        end
        predefMotion=false
    end

    --checking if there is a command from TCP_server script
    cmd=simGetStringSignal('cmdSignal')


    if (buttonID >=20000 and buttonID<=20016) or cmd~=nil then
        predefMotion=true

        --some signals cleanup and settings
        simClearStringSignal('cmdSignal')

        -------- predefined position definition
        if buttonID==20000 or cmd=='fingerPoint' then
            sequence='fingerPoint'
        end
        if buttonID==20001 or cmd=='activeIndexGrip'then
            sequence='activeIndexGrip'
        end

        if buttonID==20006 or cmd=='mouseGrip'then
            sequence='perfect'
        end
        if buttonID==20007 or cmd=='openPalmGrip' then
            sequence='openPalmGrip'
        end

        if buttonID==20009 or cmd=='powerGrip'then
            sequence='powerGrip'
        end

        if buttonID==20014 or cmd=='posicionInicial'then
            sequence='posicionInicial'
        end


    if predefMotion==true and sequence ~=nil then -- we need to handle an action script
    end
        nonpredefMotion=false --deactivate non predef motions

        h=simGetObjectHandle(sequence)
        scriptHandle=simGetScriptAssociatedWithObject(h)
        if scriptHandle==nil or h==-1 then
            simAddStatusbarMessage('script '..sequence..' does not exist')
        else
            if oldsequence~=nil and sequence~=oldsequence then
                simSetStringSignal(oldsequence,'quit')
            end
            --er=simHandleChildScript(scriptHandle)
            simAddStatusbarMessage(scriptHandle..' '..simGetScriptName(scriptHandle))

            simWriteCustomDataBlock(mainHandle,'activatedMode',simGetScriptName(scriptHandle))


            oldsequence=sequence
```

```
            end
            predefMotion=false
        end

    end
```

Anexo 3: Script "ActivateIndexGrip"

```
simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()

scriptName=simGetScriptName(sim_handle_self)
quitting=false
--print(simGetSystemTime(),'########## FIRST TIME '..scriptName)

--#######
--definition of the sequence
-- txxxx is <0: keep the current position for this finger
-- if tempo[x] is -1: subsequence will exit upon a quit or continue upon a next
-- tempo[x] is in seconds
-- pnext[x] : pointer on the index of the next move
phase={    tempo=   {0.8 ,0.5 ,-1  ,0.5 }, -- timeout on actions if -1 then no timeout but
trigger action
        pnext=     {2   ,3   ,4   ,3   }, -- pointer to the next index
        twrist=  {0 ,0 ,0 ,0 },
        tthumb=  {0   ,0   ,416 ,416 },
        tindex=  {0   ,0   ,0   ,600 },
        tmiddle= {0   ,600 ,600 ,600 },
        tring=   {0   ,600 ,600 ,600 },
        tpinky=  {0   ,600 ,600 ,600 }
        }

--sequence intitialization
index=1
indexNum=#phase.tempo
starttime=simGetSystemTime() --start action timer

while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        --print('nextaction,index',nextaction,index)
                    index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false
        print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end

        else quitting=true --no more things to do in the sequence
        end

       else
           simSwitchThread()
       end


   end
```

Anexo 4: Script "fingerPoint"

```
simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()

scriptName=simGetScriptName(sim_handle_self)
quitting=false
--print(simGetSystemTime(),'########### FIRST TIME '..scriptName)

phase={    tempo=   {0.8 }, -- timeout on actions if -1 then no timeout but trigger action
        pnext=     {2   }, -- pointer to the next index
        twrist=  {0   },
        tthumb=  {416},
        tindex=  {100},
        tmiddle= {973 },
        tring=   {973 },
        tpinky=  {973 }
        }
--sequence intitialization
index=1
indexNum=#phase.tempo
starttime=simGetSystemTime() --start action timer

while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        --print('nextaction,index',nextaction,index)
                    index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false


        print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end

        else quitting=true  --no more things to do in the sequence
        end

    else
        simSwitchThread()
    end
```

Anexo 5: Script "perfect"

```
simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()

scriptName=simGetScriptName(sim_handle_self)
quitting=false
phase={    tempo=   {0.8 }, -- timeout on actions if -1 then no timeout but trigger action
        pnext=     {2   }, -- pointer to the next index
        twrist=  {0   },
        tthumb=  {416},
        tindex=  {550 },
        tmiddle= {0  },
        tring=   {0  },
        tpinky=  {0}
        }

--sequence intitialization
index=1
indexNum=#phase.tempo
starttime=simGetSystemTime() --start action timer

while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        --print('nextaction,index',nextaction,index)
                    index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false
        print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end

                  index=phase.pnext[index]
              end
              starttime=simGetSystemTime()
            end


         else quitting=true --no more things to do in the sequence
         end

    else

        simSwitchThread()
    end

end
```

Anexo 6: Script "openPalmGrip"

```
imSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()

scriptName=simGetScriptName(sim_handle_self)
quitting=false
--print(simGetSystemTime(),'########### FIRST TIME '..scriptName)

--#######
--definition of the sequence
-- txxxx is <0: keep the current position for this finger
-- if tempo[x] is -1: subsequence will exit upon a quit or continue upon a next
-- tempo[x] is in seconds
-- pnext[x] : pointer on the index of the next move
phase={    tempo=   {1.5 }, -- timeout on actions if -1 then no timeout but trigger action
        pnext=     {2   }, -- pointer to the next index
        twrist=  {0},
        tthumb=  {90  },
        tindex=  {90  },
        tmiddle= {90  },
        tring=   {90  },
        tpinky=  {90  }
        }

--sequence intitialization
index=1
indexNum=#phase.tempo
starttime=simGetSystemTime() --start action timer

while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false
        print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end
          else quitting=true --no more things to do in the sequence
          end

       else

           simSwitchThread()
       end

   end
```

Anexo 7: Script "Posicioninicial"

```
simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()

scriptName=simGetScriptName(sim_handle_self)
quitting=false
--print(simGetSystemTime(),'########### FIRST TIME '..scriptName)

--#######
--definition of the sequence
-- txxxx is <0: keep the current position for this finger
-- if tempo[x] is -1: subsequence will exit upon a quit or continue upon a next
-- tempo[x] is in seconds
-- pnext[x] : pointer on the index of the next move
phase={    tempo=   {0.8 }, -- timeout on actions if -1 then no timeout but trigger action
        pnext=     {2    }, -- pointer to the next index
        twrist=  {0    },
        tthumb=  {0},
        tindex=  {0 },
        tmiddle= {0 },
        tring=   {0 },
        tpinky=  {0 }
        }

--sequence intitialization
index=1
indexNum=#phase.tempo
starttime=simGetSystemTime() --start action timer

while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        --print('nextaction,index',nextaction,index)
                    index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false
        print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end

        else quitting=true --no more things to do in the sequence
        end

    else
        simSwitchThread()
    end

end
```

Anexo 8: Script "PowerGrip"

```
simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()

scriptName=simGetScriptName(sim_handle_self)
quitting=false
--print(simGetSystemTime(),'########## FIRST TIME '..scriptName)

--#######
--definition of the sequence
-- txxxx is <0: keep the current position for this finger
-- if tempo[x] is -1: subsequence will exit upon a quit or continue upon a next
-- tempo[x] is in seconds
-- pnext[x] : pointer on the index of the next move
phase={    tempo=   {0.4 ,0.8 }, -- timeout on actions if -1 then no timeout but trigger
action
        pnext=     {2   ,3   }, -- pointer to the next index
        twrist=  {0 },
        tthumb=   {-1   ,416 },
        tindex=   {973 ,973 },
        tmiddle= {973 ,973 },
        tring=    {973 ,973 },
        tpinky=  {973 ,973 }
        }

--sequence intitialization
index=1
indexNum=#phase.tempo
starttime=simGetSystemTime() --start action timer

while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        --print('nextaction,index',nextaction,index)
                    index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false
        print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end

        else quitting=true --no more things to do in the sequence
        end
        simSwitchThread()
    end

end
```

Anexo 9: Script "Posiciondeterminada"

```
    simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
    simDelegateChildScriptExecution()

    scriptName=simGetScriptName(sim_handle_self)
    quitting=false
--[[fase=function(inInts,inFloats,inStrings,inBuffer)
            posicion_pulgar=inBuffer[1]
            posicion_indice=inBuffer[2]
            posicion_medio=inBuffer[3]
            posicion_anular=inBuffer[4]
            posicion_menique=inBuffer[5]
            return{},{},{},''
end]]--
-----------------------------------------------------------------------
local signal_array1=simGetStringSignal('transmision1')

if signal_array1 then
    posicion_dedos=simUnpackInts(signal_array1)
end
if posicion_dedos~=nil then
            posicion_pulgar=posicion_dedos[1]
            posicion_indice=posicion_dedos[2]
            posicion_medio=posicion_dedos[3]
            posicion_anular=posicion_dedos[4]
            posicion_menique=posicion_dedos[5]
end

    ---------------------------------------------------------------------
    phase={    tempo=   {0.8 }, -- timeout on actions if -1 then no timeout but trigger
    action
            pnext=     {2    }, -- pointer to the next index
            twrist=  {0},
            tthumb=  {posicion_pulgar},
            tindex=  {posicion_indice},
            tmiddle= {posicion_medio },
            tring=   {posicion_anular },
            tpinky=  {posicion_menique}
            }
    --sequence intitialization
    index=1
    indexNum=#phase.tempo
    starttime=simGetSystemTime() --start action timer

    while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and
    quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        --print('nextaction,index',nextaction,index)
                    index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
```

```
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false
        print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end

        else quitting=true --no more things to do in the sequence
        end

    else
        simSwitchThread()
    end

end
```

Anexo 10: Script "powerGrip"

```
simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching
simDelegateChildScriptExecution()

scriptName=simGetScriptName(sim_handle_self)
quitting=false
--print(simGetSystemTime(),'########## FIRST TIME '..scriptName)

--#######
--definition of the sequence
-- txxxx is <0: keep the current position for this finger
-- if tempo[x] is -1: subsequence will exit upon a quit or continue upon a next
-- tempo[x] is in seconds
-- pnext[x] : pointer on the index of the next move
phase={    tempo=   {0.4 ,0.8 }, -- timeout on actions if -1 then no timeout but trigger
action
        pnext=     {2    ,3    }, -- pointer to the next index
        twrist=  {0  },
        tthumb=   {-1    ,416 },
        tindex=    {973 ,973 },
        tmiddle= {973  ,973  },
        tring=     {973 ,973 },
        tpinky=  {973 ,973 }
        }

--sequence intitialization
index=1
indexNum=#phase.tempo
starttime=simGetSystemTime() --start action timer

while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) and quitting==false do
    if simReadCustomDataBlock(simGetObjectHandle('main'),'activatedMode')==scriptName then
        --#######


        data=simPackInts({phase.twrist[index],phase.tthumb[index],phase.tindex[index],phase.tm
        iddle[index],phase.tring[index],phase.tpinky[index]})

        if data~=oldata and data~=nil then
            simSetStringSignal('actuate',data)
            oldata=data
        end


        if phase.tempo[index]~=nil then
            if phase.tempo[index]>=0 then
                if simGetSystemTime()-starttime> phase.tempo[index] or nextaction=='true'
                then
        --print('nextaction,index',nextaction,index)
                    index=phase.pnext[index]
                    starttime=simGetSystemTime()
                end
            else
                if nextaction==true then
                    index=index+1
                    nextaction=false

                    --setTarget(phase.twrist[index],phase.tthumb[index],phase.tindex[index],ph
                    ase.tmiddle[index],phase.tring[index],phase.tpinky[index])
                end

                if trigger==true then
                    trigger=false
            print('------------>triggering')
                    index=phase.pnext[index]
                end
                starttime=simGetSystemTime()
            end

        else quitting=true --no more things to do  in the sequence
        end

      simSwitchThread()
    end

end
```

Anexo 11: Función "llamada_Funcion"

```
function llamada_Funcion(accion,buffer)
    ini_robotics; %Peter Corke library call
    disp('Program started');
    % vrep=remApi('remoteApi','extApi.h'); % using the header
(requires a compiler)
    vrep=remApi('remoteApi'); % using the prototype file
(remoteApiProto.m)
    vrep.simxFinish(-1); % just in case, close all opened connections
    clientID=vrep.simxStart('127.0.0.1',19999,true,true,5000,5);

    if (clientID>-1)
        disp('Connected to remote API server');


        tam=length(buffer);
        if tam~=0
            %set_position(buffer)
            posiciones=buffer;

            pos1=vrep.simxPackInts(posiciones);


vrep.simxSetStringSignal(clientID,'transmision1',pos1,vrep.simx_opmode
_streaming);

        end
        [res retInts retFloats retStrings
retBuffer]=vrep.simxCallScriptFunction(clientID,'main',vrep.sim_script
type_childscript,'displayText_function',[],[],accion,buffer,vrep.simx_
opmode_blocking);
        if (res==vrep.simx_return_ok)
            fprintf('Returned message: %s\n',retStrings);
        else
            fprintf('Remote function call failed\n');
        end
        % Now close the connection to V-REP:
        vrep.simxFinish(clientID);
    else
        disp('Failed connecting to remote API server');
    end
    vrep.delete(); % call the destructor!

    disp('Program ended');
end
```

Anexo 12: Funciones Peter Corke

VREP

V-REP simulator communications object

A VREP object holds all information related to the state of a connection to an instance of the V-REP simulator running on this or a networked computer. Allows the creation of references to other objects/models in V-REP which can be manipulated in MATLAB.

This class handles the interface to the simulator and low-level object handle operations.

Methods throw exception if an error occurs.

4

Methods

gethandle	get handle to named object
getchildren	get children belonging to handle
getobjname	get names of objects
object	return a VREP_obj object for named object
arm	return a VREP_arm object for named robot
camera	return a VREP_camera object for named vosion sensor
hokuyo	return a VREP_hokuyo object for named Hokuyo scanner
getpos	return position of object given handle
setpos	set position of object given handle
getorient	return orientation of object given handle
setorient	set orientation of object given handle
getpose	return pose of object given handle
setpose	set pose of object given handle
setobjparam_bool	set object boolean parameter
setobjparam_int	set object integer parameter
setobjparam_float	set object float parameter
getobjparam_bool	get object boolean parameter
getobjparam_int	get object integer parameter
getobjparam_float	get object float parameter
signal_int	send named integer signal
signal_float	send named float signal
signal_str	send named string signal
setparam_bool	set simulator boolean parameter
setparam_int	set simulator integer parameter
setparam_str	set simulator string parameter
setparam_float	set simulator float parameter
getparam_bool	get simulator boolean parameter
getparam_int	get simulator integer parameter
getparam_str	get simulator string parameter
getparam_float	get simulator float parameter
delete	shutdown the connection and cleanup
simstart	start the simulator running
simstop	stop the simulator running
simpause	pause the simulator
getversion	get V-REP version number
checkcomms	return status of connection
pausecomms	pause the comms
loadscene	load a scene file
clearscene	clear the current scene
loadmodel	load a model into current scene
display	print the link parameters in human readable form
char	convert to string

See also

VREP_obj, VREP_arm, VREP_camera, VREP_hokuyo

VREP.VREP

VREP object constructor

v = **VREP**(**options**) create a connection to an instance of the V-REP simulator.

Options

'timeout', T	Timeout T in ms (default 2000)
'cycle', C	Cycle time C in ms (default 5)
'port', P	Override communications port
'reconnect'	Reconnect on error (default noreconnect)
'path', P	The path to VREP install directory

Notes

- The default path is taken from the environment variable VREP

VREP.arm

Return VREP_arm object

V.**arm**(**name**) is a factory method that returns a VREP_arm object for the V-REP robot object named NAME.

Example

```
vrep.arm('IRB 140');
```

See also

VREP_arm

VREP.camera

Return VREP_camera object

V.**camera**(**name**) is a factory method that returns a VREP_camera object for the V-REP vision sensor object named NAME.

See also

VREP_camera

VREP.char

Convert to string

V.**char**() is a string representation the **VREP** parameters in human readable foramt.

See also

VREP.display

VREP.checkcomms

Check communications to V-REP simulator

V.**checkcomms**() is true if a valid connection to the V-REP simulator exists.

VREP.clearscene

Clear current scene in the V-REP simulator

V.**clearscene**() clears the current scene and switches to another open scene, if none, a new (default) scene is created.

See also

VREP.loadscene

VREP.delete

VREP object destructor

delete(**v**) closes the connection to the V-REP simulator

VREP.display

Display parameters

V.**display**() displays the **VREP** parameters in compact format.

Notes

- This method is invoked implicitly at the command line when the result of an expression is a VREP object and the command has no trailing semicolon.

See also

VREP.char

VREP.getchildren

Find children of object

C = V.**getchildren**(**H**) is a vector of integer handles for the children of the V-REP object denoted by the integer handle **H**.

VREP.gethandle

Return handle to VREP object

H = V.**gethandle**(**name**) is an integer handle for named V-REP object.

H = V.**gethandle**(**fmt**, **arglist**) as above but the name is formed from sprintf(**fmt**, **arglist**).

sprintf

VREP.getjoint

Get value of V-REP joint object

V.**getjoint**(**H**, **q**) is the position of joint object with integer handle **H**.

VREP.getobjname

Find names of objects

V.**getobjname**() will display the names and object handle (integers) for all objects in the current scene.

name = V.**getobjname**(**H**) will return the name of the object with handle **H**.

VREP.getobjparam_bool

Get boolean parameter of a V-REP object

V.**getobjparam_bool**(**H**, **param**) gets the boolean parameter with identifier **param** of object with integer handle **H**.

VREP.getobjparam_float

Get float parameter of a V-REP object

V.**getobjparam_float**(**H**, **param**) gets the float parameter with identifier **param** of object with integer handle **H**.

VREP.getobjparam_int

Get integer parameter of a V-REP object

V.**getobjparam_int**(**H**, **param**) gets the integer parameter with identifier **param** of object with integer handle **H**.

VREP.getorient

Get orientation of V-REP object

R = V.**getorient**(**H**) is the orientation of the V-REP object with integer handle **H** as a rotation matrix (3×3).

EUL = V.**getorient**(**H**, 'euler', OPTIONS) as above but returns ZYZ Euler angles.

V.**getorient**(**H**, **hrr**) as above but orientation is relative to the position of object with integer handle HR.

V.**getorient**(**H**, **hrr**, 'euler', OPTIONS) as above but returns ZYZ Euler angles.

Options

See tr2eul.

See also

VREP.setorient, VREP.getpos, VREP.getpose

VREP.getparam_bool

Get boolean parameter of the V-REP simulator

V.**getparam_bool**(**name**) is the boolean parameter with name **name** from the V-REP simulation engine.

Example

```
v = VREP();
v.getparam_bool('sim_boolparam_mirrors_enabled')
```


VREP.setparam_bool

VREP.getparam_float

Get float parameter of the V-REP simulator

V.**getparam_float**(**name**) gets the float parameter with name **name** from the V-REP simulation engine.

Example

```
v = VREP();
v.getparam_float('sim_floatparam_simulation_time_step')
```

See also

VREP.setparam_float

VREP.getparam_int

Get integer parameter of the V-REP simulator

V.**getparam_int**(**name**) is the integer parameter with name **name** from the V-REP simulation engine.

Example

```
v = VREP();
v.getparam_int('sim_intparam_settings')
```

See also

VREP.setparam_int

VREP.getparam_str

Get string parameter of the V-REP simulator

V.**getparam_str**(**name**) is the string parameter with name **name** from the V-REP simulation engine.

Example

```
v = VREP();
v.getparam_str('sim_stringparam_application_path')
```

See also

VREP.setparam_str

VREP.getpos

Get position of V-REP object

V.**getpos**(**H**) is the position (1×3) of the V-REP object with integer handle **H**.

V.**getpos**(**H**, **hr**) as above but position is relative to the position of object with integer handle **hr**.

See also

VREP.setpose, VREP.getpose, VREP.getorient

VREP.getpose

Get pose of V-REP object

T = V.**getpose**(**H**) is the pose of the V-REP object with integer handle **H** as a homogeneous transformation matrix (4×4).

T = V.**getpose**(**H**, **hr**) as above but pose is relative to the pose of object with integer handle R.

VREP.setpose, VREP.getpos, VREP.getorient

VREP.getversion

Get version of the V-REP simulator

V.**getversion**() is the version of the V-REP simulator server as an integer MNNNN where M is the major version number and NNNN is the minor version number.

VREP.hokuyo

Return VREP_hokuyo object

V.**hokuyo**(**name**) is a factory method that returns a VREP_hokuyo object for the V-REP Hokuyo laser scanner object named NAME.

See also

VREP_hokuyo

VREP.loadmodel

Load a model into the V-REP simulator

m = V.**loadmodel**(**file**, **options**) loads the model file **file** with extension .ttm into the simulator and returns a VREP_obj object that mirrors it in MATLAB.

Options

'local'	The file is loaded relative to the MATLAB client's current folder, otherwise from the V-REP root folder.

Example

```
vrep.loadmodel('people/Walking Bill');
```


Notes

- If a relative filename is given in non-local (server) mode it is relative to the V-REP models folder.

See also

VREP.arm, VREP.camera, VREP.object

VREP.loadscene

Load a scene into the V-REP simulator

V.**loadscene**(**file**, **options**) loads the scene file **file** with extension .ttt into the simulator.

Options

'local' The file is loaded relative to the MATLAB client's current folder, otherwise from the V-REP root folder.

Example

```
vrep.loadscene('2IndustrialRobots');
```

Notes

- If a relative filename is given in non-local (server) mode it is relative to the V-REP scenes folder.

See also

VREP.clearscene

VREP.mobile

Return VREP_mobile object

V.**mobile**(**name**) is a factory method that returns a VREP_mobile object for the V-REP **mobile** base object named NAME.

_mobile

VREP.object

Return VREP_obj object

V.**objet**(**name**) is a factory method that returns a VREP_obj object for the V-REP object or model named NAME.

Example

```
vrep.obj('Walking Bill');
```

See also

VREP_obj

VREP.pausecomms

Pause communcations to the V-REP simulator

V.**pausecomms**(**p**) pauses communications to the V-REP simulation engine if **p** is true else resumes it. Useful to ensure an atomic update of simulator state.

VREP.setjoint

Set value of V-REP joint object

V.**setjoint**(**H**, **q**) sets the position of joint object with integer handle **H** to the value **q**.

VREP.setjointtarget

Set target value of V-REP joint object

V.**setjointtarget**(**H**, **q**) sets the target position of joint object with integer handle **H** to the value **q**.

VREP.setjointvel

Set velocity of V-REP joint object

V.**setjointvel**(**H**, **qd**) sets the target velocity of joint object with integer handle **H** to the value **qd**.

VREP.setobjparam_bool

Set boolean parameter of a V-REP object

V.**setobjparam_bool**(**H**, **param**, **val**) sets the boolean parameter with identifier **param** of object **H** to value **val**.

VREP.setobjparam_float

Set float parameter of a V-REP object

V.**setobjparam_float**(**H**, **param**, **val**) sets the float parameter with identifier **param** of object **H** to value **val**.

VREP.setobjparam_int

Set Integer parameter of a V-REP object

V.**setobjparam_int**(**H**, **param**, **val**) sets the integer parameter with identifier **param** of object **H** to value **val**.

VREP.setorient

Set orientation of V-REP object

V.**setorient**(**H**, **R**) sets the orientation of V-REP object with integer handle **H** to that given by rotation matrix **R** (3×3).

V.**setorient**(**H**, **T**) sets the orientation of V-REP object with integer handle **H** to rotational component of homogeneous transformation matrix **T** (4×4).

V.**setorient**(**H**, **E**) sets the orientation of V-REP object with integer handle **H** to ZYZ Euler angles (1×3).

V.**setorient**(**H**, **x**, **hr**) as above but orientation is set relative to the orientation of object with integer handle **hr**.

See also

VREP.getorient, VREP.setpos, VREP.setpose

VREP.setparam_bool

Set boolean parameter of the V-REP simulator

V.**setparam_bool**(**name**, **val**) sets the boolean parameter with name **name** to value **val** within the V-REP simulation engine.

See also

VREP.getparam_bool

VREP.setparam_float

Set float parameter of the V-REP simulator

V.**setparam_float**(**name**, **val**) sets the float parameter with name **name** to value **val** within the V-REP simulation engine.

See also

VREP.getparam_float

VREP.setparam_int

Set integer parameter of the V-REP simulator

V.**setparam_int**(**name**, **val**) sets the integer parameter with name **name** to value **val** within the V-REP simulation engine.

See also

VREP.getparam_int

VREP.setparam_str

Set string parameter of the V-REP simulator

V.**setparam_str**(**name**, **val**) sets the integer parameter with name **name** to value **val** within the V-REP simulation engine.

See also

VREP.getparam_str

VREP.setpos

Set position of V-REP object

V.**setpos**(**H**, **T**) sets the position of V-REP object with integer handle **H** to **T** (1×3).

V.**setpos**(**H**, **T**, **hr**) as above but position is set relative to the position of object with integer handle **hr**.

See also

VREP.getpos, VREP.setpose, VREP.setorient

VREP.setpose

Set pose of V-REP object

V.**setpos**(**H**, **T**) sets the pose of V-REP object with integer handle **H** according to homogeneous transform **T** (4×4).

V.**setpos**(**H**, **T**, **hr**) as above but pose is set relative to the pose of object with integer handle **hr**.

See also

VREP.getpose, VREP.setpos, VREP.setorient

VREP.signal_float

Send a float signal to the V-REP simulator

V.**signal_float**(**name**, **val**) send a float signal with name **name** and value **val** to the V-REP simulation engine.

VREP.signal_int

Send an integer signal to the V-REP simulator

V.**signal_int**(**name**, **val**) send an integer signal with name **name** and value **val** to the V-REP simulation engine.

VREP.signal_str

Send a string signal to the V-REP simulator

V.**signal_str**(**name**, **val**) send a string signal with name **name** and value **val** to the V-REP simulation engine.

VREP.simpause

Pause V-REP simulation

V.**simpause**() pauses the V-REP simulation engine. Use V.simstart() to resume the simulation.

See also

VREP.simstart

VREP.simstart

Start V-REP simulation

V.**simstart**() starts the V-REP simulation engine.

See also

VREP.simstop, VREP.simpause

VREP.simstop

Stop V-REP simulation

V.**simstop**() stops the V-REP simulation engine.

See also

VREP.simstart

VREP.youbot

Return VREP_youbot object

V.**youbot**(**name**) is a factory method that returns a VREP_youbot object for the V-REP YouBot object named NAME.

See also

vrep_youbot

VREP_arm

Mirror of V-REP robot arm object

Mirror objects are MATLAB objects that reflect the state of objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This is a concrete class, derived from VREP_mirror, for all V-REP robot arm objects and allows access to joint variables.

Methods throw exception if an error occurs.

Example

```
vrep = VREP();
arm = vrep.arm('IRB140');
q = arm.getq();
arm.setq(zeros(1,6));
arm.setpose(T);  % set pose of base
```

Methods

getq	get joint coordinates
setq	set joint coordinates
setjointmode	set joint control parameters
animate	animate a joint coordinate trajectory
teach	graphical teach pendant

Superclass methods (VREP_obj)

getpos	get position of object
setpos	set position of object
getorient	get orientation of object
setorient	set orientation of object
getpose	get pose of object given
setpose	set pose of object

can be used to set/get the pose of the robot base.

Superclass methods (VREP_mirror)

getname	get object name
setparam_bool	set object boolean parameter
setparam_int	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter

Properties

n	Number of joints

See also

VREP_mirror, VREP_obj, VREP_arm, VREP_camera, VREP_hokuyo

VREP_arm.VREP_arm

Create a robot arm mirror object

arm = **VREP_arm**(**name**, **options**) is a mirror object that corresponds to the robot arm named **name** in the V-REP environment.

Options

'fmt', F	Specify format for joint object names (default '%s_joint%d')

Notes

- The number of joints is found by searching for objects with names systematically derived from the root object name, by default named NAME_N where N is the joint number starting at 0.

See also

VREP.arm

VREP_arm.animate

Animate V-REP robot

R.**animate**(**qt**, **options**) animates the corresponding V-REP robot with configurations taken from consecutive rows of **qt** ($M \times N$) which represents an M-point trajectory and N is the number of robot joints.

Options

'delay', D	Delay (s) betwen frames for animation (default 0.1)
'fps', fps	Number of frames per second for display, inverse of 'delay' option
'[no]loop'	Loop over the trajectory forever

See also

SerialLink.plot

VREP_arm.getq

Get joint angles of V-REP robot

ARM.**getq**() is the vector of joint angles ($1 \times N$) from the corresponding robot arm in the V-REP simulation.

See also

VREP_arm.setq

VREP_arm.setjointmode

Set joint mode

ARM.**setjointmode**(**m**, **C**) sets the motor enable **m** (0 or 1) and motor control **C** (0 or 1) parameters for all joints of this robot arm.

VREP_arm.setq

Set joint angles of V-REP robot

ARM.**setq**(**q**) sets the joint angles of the corresponding robot arm in the V-REP simulation to **q** ($1 \times N$).

See also

VREP_arm.getq

VREP_arm.setqt

Set joint angles of V-REP robot

ARM.**setq**(**q**) sets the joint angles of the corresponding robot arm in the V-REP simulation to **q** $(1 \times N)$.

VREP_arm.teach

Graphical teach pendant

R.**teach**(**options**) drive a V-REP robot by means of a graphical slider panel.

Options

'degrees'	Display angles in degrees (default radians)
'q0', q	Set initial joint coordinates

Notes

- The slider limits are all assumed to be [-pi, +pi]

See also

SerialLink.plot

VREP_camera

Mirror of V-REP vision sensor object

Mirror objects are MATLAB objects that reflect the state of objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This is a concrete class, derived from VREP_mirror, for all V-REP vision sensor objects and allows access to images and image parameters.

Methods throw exception if an error occurs.

Example

```
vrep = VREP();
camera = vrep.camera('Vision_sensor');
im = camera.grab();
camera.setpose(T);
R = camera.getorient();
```

Methods

grab	return an image from simulated camera
setangle	set field of view
setresolution	set image resolution
setclipping	set clipping boundaries

Superclass methods (VREP_obj)

getpos	get position of object
setpos	set position of object
getorient	get orientation of object
setorient	set orientation of object
getpose	get pose of object
setpose	set pose of object

can be used to set/get the pose of the robot base.

Superclass methods (VREP_mirror)

getname	get object name
setparam_bool	set object boolean parameter
setparam_int	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter

See also

VREP_mirror, VREP_obj, VREP_arm, VREP_camera, VREP_hokuyo

VREP_camera.VREP_camera

Create a camera mirror object

C = **VREP_camera**(**name**, **options**) is a mirror object that corresponds to the vision senor named **name** in the V-REP environment.

Options

'fov', A	Specify field of view in degreees (default 60)
'resolution', N	Specify resolution. If scalar $N \times N$ else N(1)xN(2)
'clipping', Z	Specify near Z(1) and far Z(2) clipping boundaries

Notes

- Default parameters are set in the V-REP environmen
- Can be applied to "DefaultCamera" which controls the view in the simulator GUI.

See also

VREP_obj

VREP_camera.char

Convert to string

V.**char**() is a string representation the VREP parameters in human readable foramt.

See also

VREP.display

VREP_camera.getangle

Fet field of view for V-REP vision sensor

fov = C.**getangle**(**fov**) is the field-of-view angle to **fov** in radians.

See also

VREP_camera.setangle

VREP_camera.getclipping

Get clipping boundaries for V-REP vision sensor

C.**getclipping**() is the near and far clipping boundaries (1×2) in the Z-direction as a 2-vector [NEAR,FAR].

See also

VREP_camera.setclipping

VREP_camera.getresolution

Get resolution for V-REP vision sensor

R = C.**getresolution**() is the image resolution (1×2) of the vision sensor **R**(1)xR(2).

See also

VREP_camera.setresolution

VREP_camera.grab

Get image from V-REP vision sensor

im = C.**grab**(**options**) is an image $(W \times H)$ returned from the V-REP vision sensor.

C.**grab**(**options**) as above but the image is displayed using idisp.

Options

'grey' Return a greyscale image (default color).

V-REP simulator must be running.

- Color images can be quite dark, ensure good light sources.
- Uses the signal 'handle_rgb_sensor' to trigger a single image generation.

See also

idisp, VREP.simstart

VREP_camera.setangle

Set field of view for V-REP vision sensor

C.**setangle**(**fov**) set the field-of-view angle to **fov** in radians.

See also

VREP_camera.getangle

VREP_camera.setclipping

Set clipping boundaries for V-REP vision sensor

C.**setclipping**(**near**, **far**) set clipping boundaries to the range of Z from **near** to **far**. Objects outside this range will not be rendered.

See also

VREP_camera.getclipping

VREP_camera.setresolution

Set resolution for V-REP vision sensor

C.**setresolution**(**R**) set image resolution to $\mathbf{R} \times \mathbf{R}$ if **R** is a scalar or **R**(1)xR(2) if it is a 2-vector.

By default V-REP cameras seem to have very low (32×32) resolution.

- Frame rate will decrease as frame size increases.

See also

VREP_camera.getresolution

VREP_mirror

V-REP mirror object class

Mirror objects are MATLAB objects that reflect the state of objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This abstract class is the root class for all V-REP mirror objects.

Methods throw exception if an error occurs.

Methods

getname	get object name
setparam_bool	set object boolean parameter
setparam_int	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter
remove	remove object from scene
display	display object info
char	convert to string

Properties (read only)

h	V-REP integer handle for the object
name	Name of the object in V-REP
vrep	Reference to the V-REP connection object

This has nothing to do with mirror objects in V-REP itself which are shiny reflective surfaces.

See also

VREP_obj, VREP_arm, VREP_camera, VREP_hokuyo

VREP_mirror.VREP_mirror

Construct VREP_mirror object

obj = **VREP_mirror**(**name**) is a V-REP mirror object that represents the object named **name** in the V-REP simulator.

VREP_mirror.char

Convert to string

OBJ.**char**() is a string representation the VREP parameters in human readable foramt.

See also

VREP.display

VREP_mirror.display

Display parameters

OBJ.**display**() displays the VREP parameters in compact format.

Notes

- This method is invoked implicitly at the command line when the result of an expression is a VREP object and the command has no trailing semicolon.

See also

VREP.char

VREP_mirror.getname

Get object name

OBJ.**getname**() is the name of the object in the VREP simulator.

VREP_mirror.getparam_bool

Get boolean parameter of V-REP object

OBJ.**getparam_bool**(**id**) is the boolean parameter with **id** of the corresponding V-REP object.

See also **VREP_mirror**.setparam_bool, **VREP_mirror**.getparam_int, **VREP_mirror**.getparam_float.

VREP_mirror.getparam_float

Get float parameter of V-REP object

OBJ.**getparam_float**(**id**) is the float parameter with **id** of the corresponding V-REP object.

See also **VREP_mirror**.setparam_bool, **VREP_mirror**.getparam_bool, **VREP_mirror**.getparam_int.

VREP_mirror.getparam_int

Get integer parameter of V-REP object

OBJ.**getparam_int**(**id**) is the integer parameter with **id** of the corresponding V-REP object.

See also **VREP_mirror**.setparam_int, **VREP_mirror**.getparam_bool, **VREP_mirror**.getparam_float.

VREP_mirror.setparam_bool

Set boolean parameter of V-REP object

OBJ.**setparam_bool**(**id**, **val**) sets the boolean parameter with **id** to value **val** within the V-REP simulation engine.

See also **VREP_mirror**.getparam_bool, **VREP_mirror**.setparam_int, **VREP_mirror**.setparam_float.

VREP_mirror.setparam_float

Set float parameter of V-REP object

OBJ.**setparam_float**(**id**, **val**) sets the float parameter with **id** to value **val** within the V-REP simulation engine.

See also **VREP_mirror**.getparam_float, **VREP_mirror**.setparam_bool, **VREP_mirror**.setparam_int.

VREP_mirror.setparam_int

Set integer parameter of V-REP object

OBJ.**setparam_int**(**id**, **val**) sets the integer parameter with **id** to value **val** within the V-REP simulation engine.

See also **VREP_mirror**.getparam_int, **VREP_mirror**.setparam_bool, **VREP_mirror**.setparam_float.

VREP_obj

V-REP mirror of simple object

Mirror objects are MATLAB objects that reflect objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This is a concrete class, derived from VREP_mirror, for all V-REP objects and allows access to pose and object parameters.

Example

```
vrep = VREP();
bill = vrep.object('Bill');  % get the human figure Bill
bill.setpos([1,2,0]);
bill.setorient([0 pi/2 0]);
```

Methods throw exception if an error occurs.

Methods

getpos	get position of object
setpos	set position of object
getorient	get orientation of object
setorient	set orientation of object
getpose	get pose of object
setpose	set pose of object

Superclass methods (VREP_mirror)

getname	get object name
setparam_bool	set object boolean parameter
setparam_int	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter
display	print the link parameters in human readable form
char	convert to string

See also

VREP_mirror, VREP_obj, VREP_arm, VREP_camera, VREP_hokuyo

VREP_obj.VREP_obj

VREP_obj mirror object constructor

v = **VREP_base**(**name**) creates a V-REP mirror object for a simple V-REP object type.

VREP_obj.getorient

Get orientation of V-REP object

V.**getorient**() is the orientation of the corresponding V-REP object as a rotation matrix (3×3).

V.**getorient**('euler', OPTIONS) as above but returns ZYZ Euler angles.

V.**getorient**(**base**) is the orientation of the corresponding V-REP object relative to the **VREP_obj** object **base**.

V.**getorient**(**base**, 'euler', OPTIONS) as above but returns ZYZ Euler angles.

Options

See tr2eul.

See also

VREP_obj.setorient, VREP_obj.getopos, VREP_obj.getpose

VREP_obj.getpos

Get position of V-REP object

V.**getpos**() is the position (1×3) of the corresponding V-REP object.

V.**getpos**(**base**) as above but position is relative to the **VREP_obj** object **base**.

See also

VREP_obj.setpos, VREP_obj.getorient, VREP_obj.getpose

VREP_obj.getpose

Get pose of V-REP object

V.**getpose**() is the pose (4×4) of the the corresponding V-REP object.

V.**getpose**(**base**) as above but pose is relative to the pose the **VREP_obj** object **base**.

See also

VREP_obj.setpose, VREP_obj.getorient, VREP_obj.getpos

VREP_obj.setorient

Set orientation of V-REP object

V.**setorient**(**R**) sets the orientation of the corresponding V-REP to rotation matrix **R** (3×3).

V.**setorient**(**T**) sets the orientation of the corresponding V-REP object to rotational component of homogeneous transformation matrix **T** (4×4).

V.**setorient**(**E**) sets the orientation of the corresponding V-REP object to ZYZ Euler angles (1×3).

V.**setorient**(**x**, **base**) as above but orientation is set relative to the orientation of **VREP_obj** object **base**.

See also

VREP_obj.getorient, VREP_obj.setpos, VREP_obj.setpose

VREP_obj.setpos

Set position of V-REP object

V.**setpos**(**T**) sets the position of the corresponding V-REP object to **T** (1×3).

V.**setpos**(**T**, **base**) as above but position is set relative to the position of the **VREP_obj** object **base**.

See also

VREP_obj.getpos, VREP_obj.setorient, VREP_obj.setpose

VREP_obj.setpose

Set pose of V-REP object

V.**setpose**(**T**) sets the pose of the corresponding V-REP object to **T** (4×4).

Universidad de Alcalá

Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR